A Methodology to Develop High Performance Applications on GPGPU Architectures: Application to Simulation of Electrical Machines

de Oliveira Rodrigues Antonio Wendell

▶ **To cite this version:**

de Oliveira Rodrigues Antonio Wendell. A Methodology to Develop High Performance Applications on GPGPU Architectures: Application to Simulation of Electrical Machines. Electromagnetism. Université des Sciences et Technologie de Lille - Lille I, 2012. English. NNT : . tel-00670221

HAL Id: tel-00670221

https://theses.hal.science/tel-00670221

Submitted on 14 Feb 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.






Numéro d'ordre: 40762

Université des Sciences et Technologies de Lille
École Doctorale Sciences pour l'Ingénieur

THÈSE

présentée pour obtenir le titre de docteur
spécialité Informatique

par

ANTONIO WENDELL DE OLIVEIRA RODRIGUES

UNE MÉTHODOLOGIE POUR LE DÉVELOPPEMENT D'APPLICATIONS HAUTES PERFORMANCES SUR DES ARCHITECTURES GPGPU: APPLICATION À LA SIMULATION DES MACHINES ÉLÉCTRIQUES

Thèse soutenue le 26 Janvier 2012, devant la commission d'examen formée de :

1	Pierre Manneback	Rapporteur/Président
2	Sven-Bodo Scholz	Rapporteur
3	Yvonnick Le Menach	Examinateur
4	Mamy Rakotovao	Examinateur
5	Frédéric Guyomarc'h	Co-Directeur
6	Jean-Luc Dekeyser	Directeur

Université des Sciences et Technologies de Lille
LIFL - Cité Scientifique, Bat. M3 - 59655 Villeneuve d'Ascq Cedex






Number : 40762

Université des Sciences et Technologies de Lille
École Doctorale Sciences pour l'Ingérnieur

THESIS

submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science

by

ANTONIO WENDELL DE OLIVEIRA RODRIGUES

A METHODOLOGY TO DEVELOP HIGH PERFORMANCE APPLICATIONS ON GPGPU ARCHITECTURES: APPLICATION TO SIMULATION OF ELECTRICAL MACHINES

January 26, 2012
Commitee in charge

1	Pierre Manneback	Reviewer/President
2	Sven-Bodo Scholz	Reviewer
3	Yvonnick Le Menach	Examiner
4	Mamy Rakotovao	Examiner
4	Frédéric Guyomarc'h	Co-advisor
6	Jean-Luc Dekeyser	Advisor

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE
LIFL - Cité Scientifique, Bat. M3 - 59655 Villeneuve d'Ascq Cedex

A METHODOLOGY TO DEVELOP HIGH PERFORMANCE APPLICATIONS ON GPGPU ARCHITECTURES: APPLICATION TO SIMULATION OF ELECTRICAL MACHINES

ANTONIO WENDELL DE OLIVEIRA RODRIGUES

Doctorate Thesis
January 2012

Antonio Wendell de Oliveira Rodrigues: *A Methodology to Develop High Performance Applications on GPGPU Architectures: Application to Simulation of Electrical Machines,* Doctorate Thesis © January 2012

This thesis is dedicated to my parents.

For their endless love, support and encouragement

RESUMÉ

Les phénomènes physiques complexes peuvent être simulés numériquement par des techniques mathématiques basées souvent sur la discrétisation des équations aux dérivées partielles régissant ces phénomènes. Ces simulations peuvent mener ainsi à la résolution de très grands systèmes. La parallélisation des codes de simulation numérique, c'est-à-dire leur adaptation aux architectures des calculateurs parallèles, est alors une nécessité pour parvenir à faire ces simulations en des temps non-exorbitants. Le parallélisme s'est imposé au niveau des architectures de processeurs et les cartes graphiques sont maintenant utilisées pour des fins de calcul généraliste, aussi appelé "General-Purpose computation on Graphics Processing Unit (GPGPU)", avec comme avantage évident l'excellent rapport performance/prix.

Cette thèse se place dans le domaine de la conception de ces applications hautes-performances pour la simulation des machines électriques. Nous fournissons une méthodologie basée sur l'Ingénierie Dirigées par les Modèles (IDM) qui permet de modéliser une application et l'architecture sur laquelle l'exécuter afin de générer un code OpenCL. Notre objectif est d'aider les spécialistes en algorithmes de simulations numériques à créer un code efficace qui tourne sur les architectures GPGPU. Pour cela, une chaine de compilation de modèles qui prend en compte les plusieurs aspects du modèle de programmation OpenCL est fournie. De plus, pour rendre le code raisonnablement efficace par rapport à un code développé à la main, nous fournissons des transformations de modèles qui regardent des niveaux d'optimisations basées sur les caractéristiques de l'architecture (niveau de mémoire par exemple).

Comme validation expérimentale, la méthodologie est appliquée à la création d'une application qui résout un système linéaire issu de la Méthode des Éléments Finis pour la simulation de machines électriques. Dans ce cas nous montrons, entre autres, la capacité de la méthodologie de passer à l'échelle par une simple modification de la multiplicité des unités GPU disponibles.

Mots-clés: MDE, UML, MARTE, Transformation de Modéles, Génération Automatique de Code, GPGPU, OpenCL, Simulation Numérique, Machines Électriques

ABSTRACT

Complex physical phenomena can be numerically simulated by mathematical techniques. Usually, these techniques are based on discretization of partial differential equations that govern these phenomena. Hence, these simulations enable the solution of large-scale systems. The parallelization of algorithms of numerical simulation, i. e., their adaptation to parallel processing architectures, is an aim to reach in order to hinder exorbitant execution times. The parallelism has been imposed at the level of processor architectures and graphics cards are now used for purposes of general calculation, also known as "General-Purpose computation on Graphics Processing Unit (GPGPU)". The clear benefit is the excellent performance/price ratio.

This thesis addresses the design of high-performance applications for simulation of electrical machines. We provide a methodology based on Model Driven Engineering (MDE) to model an application and its execution architecture in order to generate OpenCL code. Our goal is to assist specialists in algorithms of numerical simulations to create a code that runs efficiently on GPGPU architectures. To ensure this, we offer a compilation model chain that takes into account several aspects of the OpenCL programming model. In addition, to get a code fairly efficient compared to a code developed manually, we provide model transformations that analyze some levels of optimizations based on the characteristics of the architecture (e. g. memory issues).

As an experimental validation, the methodology is applied to the creation of an application that solves a linear system resulting from the Finite Element Method (FEM) for simulation of electrical machines. In this case, we show, among other things, the ability of the methodology of scaling by a simple modification of the number of available GPU devices.

Keywords: MDE, UML, MARTE, Model Transformation, Automatic Code Generation, GPGPU, OpenCL, Numerical Simulation, Electrical Machines

PUBLICATIONS

Some proposals and figures have appeared previously in the following publications:

JOURNALS

1. Antonio Wendell De Oliveira Rodrigues, Frédéric Guyomarc'h, Yvonnick Le Menach, and Jean-Luc Dekeyser. Automatic Multi-GPU Code Generation applied to Simulation of Electrical Machines. *Magnetics, IEEE Transactions on*, 48(2):831 –834, Feb. 2012. ISSN 0018-9464. doi: 10.1109/TMAG.2011.2179527
2. Antonio Wendell De Oliveira Rodrigues, Frédéric Guyomarc'h, and Jean-Luc Dekeyser. An MDE Approach for Automatic Code Generation from UML/MARTE to OpenCL. *IEEE Computer in Science & Engineering - Special Edition on GPUs, Journal*, Jan 2012 (to appear)

CONFERENCES AND WORKSHOPS

1. Antonio Wendell De Oliveira Rodrigues, Frédéric Guyomarc'h, Yvonnick Le Menach, and Jean-Luc Dekeyser. Parallel Sparse Matrix Solver on the GPU Applied to Simulation of Electrical Machines. In *Compumag 2009*, Florianopolis, Brazil, November 2009
2. Antonio Wendell De Oliveira Rodrigues, Frédéric Guyomarc'h, and Jean-Luc Dekeyser. Programming Massively Parallel Architectures using MARTE: a Case Study. In *2nd Workshop on Model Based Engineering for Embedded Systems Design (M-BED) on Date Conference 2011*, Grenoble, France, March 2011
3. Antonio Wendell De Oliveira Rodrigues, Frédéric Guyomarc'h, and Jean-Luc Dekeyser. Using ArrayOL to Identify Potentially Shareable Data in Thread Work-Groups of GPUs. In *Designing for Embedded Parallel Computing Platforms: Architectures, Design Tools, and Applications on DATE 2011*, Grenoble, France, March 2011. Work in-Progress Poster
4. Antonio Wendell De Oliveira Rodrigues, Frédéric Guyomarc'h, and Jean-Luc Dekeyser. A Modeling Approach based on UML/MARTE for GPU Architecture. In *Symposium en Architectures nouvelles de machines (SympA'14)*, Saint Malo, France, May 2011

5. Jing Guo, Antonio Wendell De Oliveira Rodrigues, Jerarajan Thiyagalingam, Frédéric Guyomarc'h, Pierre Boulet, and Sven-Bodo Scholz. Harnessing the Power of GPUs without Losing Abstractions in SaC and ArrayOL: A Comparative Study. In *HIPS 2011, 16th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Anchorage (Alaska), United States of America, May 2011
6. Antonio Wendell De Oliveira Rodrigues, Frédéric Guyomarc'h, Jean-Luc Dekeyser, and Yvonnick Le Menach. Automatic Multi-GPU Code Generation applied to Simulation of Electrical Machines. In *Compumag 2011*, Sydney, Australia, July 2011

RESEARCH REPORT

1. Antonio Wendell De Oliveira Rodrigues, Frédéric Guyomarc'h, and Jean-Luc Dekeyser. An MDE Approach for Automatic Code Generation from MARTE to OpenCL. Research Report RR-7525, INRIA, February 2011. Research Report RR-7525 in http://hal.inria.fr/inria-00563411
2. Antonio Wendell De Oliveira Rodrigues, Vincent Aranega, Anne Etien, Frédéric Guyomarc'h, and Jean-Luc Dekeyser. Enabling Traceability in an MDE Approach to Improve Performance of GPU Applications. Rapport de recherche RR-7720, INRIA, August 2011. Research Report RR-7720 in http://hal.inria.fr/inria-00617912

ACKNOWLEDGMENTS

First and foremost, I have to thank my father, Antonio, and specially my beloved mother, Francisca, for their love and support throughout my life. Thank you both for giving me strength to reach for the stars and chase my dreams. My sisters, Weidina, Weidiany, and Weidinara, deserve my wholehearted thanks as well. To my dearly loved fiancée (and future wife) Kamila, thank you for your assistance, your encouragement, and your company (even if sometimes virtual). I am very happy by your side. I owe this thesis to all of you.

I would like to sincerely thank my supervisors, Jean-Luc Dekeyser and Frédéric Guyomarc'h (the best co-advisor ever), for their guidance and support throughout this study, and especially for their confidence in me. I hope to continue working with you in the future. To all researchers from DaRT team, above all, Anne Etien, Abdoulaye Gamatié, and Pierre Boulet, people with whom I discussed many points of my work concerning their research field. I would like also to thank Alexis Muller and Thomas Legrand, for so many times that I bothered them to help me to fix some issues on the Gaspard's tools, and they kindly always helped me. To all colleagues and PhD students from the office 111bis, the former ones: Imran, Adolf, Calin, and Meriem, also the current ones: Chiraz, Sana, Amine, and Pamela, my sincere thanks. Furthermore, I would like to express my gratitude to Karine Lewandowski for her administrative support always well done. I cannot forget to thank you, Vincent Aranega, for sharing ideas during our researches, the pleasure working together, and foremost, your friendship.

This thesis relies on industrial applications. I have to thank Yvonnick Le Menach, Francis Piriou, both from L2EP laboratory, and Mamy Rakotovao from Valeo, for giving me examples, opinions, and assistance to do the experimental validation of my results.

Halfway through the journey of my thesis, I had the great opportunity to work with the SAC's team at the University of Hertfordshire. This was really valuable to my work. In particular, thanks go to Sven-Bodo, Jing Guo, Nil, Dan, and Jeyarajan.

Living in France allowed me to meet some new and old friends. Raquel and Ludovic, you do not know how important you were and are when you give me the warmest welcomes at your home, thank you. Thanks go also to you, Alexis Deneux, Joelia (I cannot get count how many times you gave me a hand), Cariza, Mariusa, and Nizar. Your friendship and support were essential to my stay. Thank you very much, Carina e Reinaldo, my former students, and now two great friends who gave me a good time in Grenoble and

Dublin. Furthermore, to Carina's father and my previous advisor, Mauro Oliveira, who has a valuable role in my life as a researcher, thanks.

I have to express my gratitude to the Federal Institute of Ceara (IFCE) and my country, Brazil. I am very grateful to all my friends from the Telematica Department for replacing me while I was absent. Thanks to my friend César Olavo, who gave me the opportunity to meet the DaRT team, and consequently to work with them. I have to mention yet some special friends that always believed in the success of this thesis. Thank you, Joesito and Bené, for being such a good "big brother/sister" to me. Thank you, Janaina, I know you a long time, and I feel our friendship growing more and more, this makes me happy. By the way, thank you for undertaking all administrative stuff at IFCE. I am very pleased to work with you again soon.

In 2011, I lost a big friend. He was directly responsible for me deciding to do a doctorate. Thank you, Valdson. My dear friend Nayara, who always asked me how my work was going, and who gave me the right words to keep my motivation, my humble thank you. Furthermore, thanks to my dear friends Corneli Jr. and Italo for being there whenever I took a break of my works and left to Brazil.

Thank you, Lord, for always being there for me.

Thanks to Valeo and Région Nord-Pas de Calais for supporting this thesis with total confidence in my works.

This thesis was written in English by a Brazilian. However, I am lucky that it had been conducted by researchers who showed me the best principles of the French research.

Finally, I would like to thank everybody who was important to the successful realization of this thesis, as well as expressing my apology that I could not mention personally one by one.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ATL	Atlas Transformation Language
ArrayOL	Array Oriented Language
BiCGCR	BiConjugate Gradient Conjugate Residual
Code_CARMEL	Code Avancé de Recherche pour les Machines Électriques
CG	Conjugate Gradient
CPU	Central Processing Unit
CSR	Compressed Sparse Row
DAG	Directed Acyclic Graph
DSL	Domain-Specific Language
EDF	Électricité de France
EMF	Eclipse Modeling Framework
FEM	Finite Element Method
FFT	Fast Fourier Transform
Gaspard2	Graphical Array Specification for Parallel and Distribute Computing
GPGPU	General-Purpose computation on Graphics Processing Unit
GPU	Graphics Processor Unit
HD-CIF	High Definition Common Intermediate Format
HPC	High Performance Computing
HPF	High Performance Fortran
HRM	Hardware Resources Modeling
IP	Intellectual Property
L2EP	Laboratory of Electrical and Power Electronics of Lille
M2M	Model-to-Model Transformation
M2T	Model-to-Text Transformation
MARTE	Modeling and Analysis of Real-Time and Embedded Systems
MDA	Model-Driven Architecture
MDE	Model-Driven Engineering
MIMD	Multiple Instruction, Multiple Data
MoC	Model of Computation

MOF	MetaObject Facility
MPI	Message Parsing Interface
MPP	Massively Parallel Processing
MUMPS	MUltifrontal Massively Parallel sparse direct Solver
PDE	Partial Differential Equations
OCL	Object Constraint Language
OMG	Object Management Group
OML	Operational Mapping Language
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
QVT	Query/View/Transformation
QVTO	Query/View/Transformation Operational
RSM	Repetitive Structure Modeling
SMP	Symmetric Multi-Processing
SIMD	Single Instruction, Multiple Data
SPMD	Single Program, Multiple Data
UML	Unified Modeling Language
XMI	XML Metadata Interchange

INTRODUCTION

CONTEXT

In the recent decades, numerical simulation has become a valuable and successful method for solving complex problems in almost all areas of physics and human sciences. Indeed, numerical simulations are a useful part of mathematical modeling of many natural systems in physics, astrophysics, economics, psychology, social science, biochemistry, and engineering.

Numerical simulation is fundamental to engineering advances specially in the electromagnetic fields. The application range extends from low frequency industrial applications up to ultra-high frequency applications. However, there are many problems that simply do not have analytical solutions, and there are those whose exact solution is beyond the current state of knowledge. Indeed, for many of these real-world electromagnetic problems like scattering, radiation, wave guiding, there is no analytical computable solution due to the complexity and multitude of irregular geometries found in actual devices. Most of these problems can not be solved manually. Therefore, developers have implemented a number of advanced numerical algorithms specifically designed for those applications. For example, numerical simulation techniques can overcome the inability to derive closed form solutions of Maxwell's equations under various constitutive relations of media, and boundary conditions.

Throughout the years, software developers have worked on the transformation of numerical algorithms into useable programs. There are several options available to developers, both in terms of language and overall approach. Often, the programming solution of numerical simulations can become as intricate and involved as the original problems and requires almost as much refinement and care to obtain a solution. This usually requires a twofold skill from developers (or developing team): knowledge of the problem's domain and the programming language. The former belongs to physicists that have an in-depth knowledge of physical phenomena and thus are able to propose suitable solutions according to the problem, e. g. mathematical interpretation of physical phenomena. The latter lies in programming languages and execution platforms, where developers are able to exploit the architecture computing power aiming high performances. In order to promote the rapid development, many numerical libraries have emerged in different programming languages, such as Fortran, Java, C/C++ [122, 103, 86]. Most of these libraries relate to linear algebra where classical functions are often used in the implementa-

tion of numerical algorithms. Moreover, there are dedicated software packages that increase developing productivity for mathematical algorithms. Packages such as Matlab [99] and Scilab [30] integrate computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation.

In numerical simulations, scientific computing programs often produce a large amount of simulation data. Consequently, this requires more computing power than provided by sequential computers. Otherwise, many of these programs can take very long time to solve a problem. Current hardware architectures offering high performance do not only exploit parallelism within a single processor via multiple CPU cores but also apply a medium to large number of processors concurrently to a single computation. However, switching from sequential approaches to parallel approaches has been a recurring issue in programming paradigm. Traditional *de facto* standards, such as Message Parsing Interface (MPI) [56] and Open Multi-Processing (OpenMP) [121], are often used as parallel approaches to implement high performance applications. Nevertheless, new parallel architectures are often proposed. Consequently, new programming models for those architectures will require new skills from developers.

ISSUES AND MOTIVATIONS

Within the context described above, we take into account some concerns related to parallel programming, emerging parallel technologies and high-level specification of numerical simulation algorithms. Under these three aspects, we emphasize the challenges still to be faced, and that we consider necessary for scientific and industrial community.

Parallel Programming Paradigm

In modern execution platforms, parallelism appears at various levels both in hardware and software: signal, circuit, component, and system levels. For instance, at a higher level, SMP systems have multiple CPUs that work in parallel. At an even higher level of parallelism, one can connect several computers together and make them work as a single machine, popularly known as cluster computing. Nevertheless, the main challenge to developers is deciding the lumps of code, i. e., the code granularity, that can be a potential candidate for parallelism. Several solutions to granularity decision process have already been proposed, such as those in [33, 67]. However, these solutions deal with idealized systems and most of the overhead parameters, e. g. communication, are ignored. Thus, we do not consider it optimal to

An overview on HPC systems and clusters is provided in Appendix A.

automatically define the code granularity. Hence, developers must undertake this process.

Matson et al. [101] identify the top 10 issues concerning parallel programming. Among these issues, we considered some major ones as essential such as:

④ *Supporting scalability, hardware: bandwidth and latencies to memory plus interconnects between processors to help applications scale.*
Although a software development methodology does not allow changing the hardware architecture, it is possible to take advantage of the available resources according to applications.

⑤ *Supporting scalability, software: libraries, scalable algorithms, and adaptive runtimes to map high-level software onto platform details.*

⑦ *Tools, API's and methodologies to support the debugging process.*
For instance, mechanisms for optimization and fine-tuning of parallel applications.

⑨ *Support for good software engineering practices: composability, incremental parallelism, and code reuse.*
Furthermore, support for a compact expression of parallelism at higher levels.

⑩ *Support for portable performance. What are the right models (or abstractions) so programmers can write code once and expect it to execute well on the important parallel platforms?*

All these issues are challenges in parallel programming. Researchers have constantly proposed solutions to bridge existing gaps.

Massively Parallel Architectures

Massively parallel architectures are computing systems that consist of many individual nodes. The term *massive* is wide and connotes hundreds if not thousands of such units. However, growth in this area has been driven by the development of graphics cards with massively parallel processors, resulting in commodity hardware that is widely available, scalable and cost-effective. The adoption of general-purpose hardware for graphics applications has opened the door for non-graphics applications to use the potential of the graphics card (or GPUs). Graphics Processor Unit (GPU)s, from this point, is our aimed platform. They have led the *many-core* performances since 2003. Particularly, we focus on the OpenCL programming and execution models. In OpenCL, there is a host system that manages one or more devices. In order to define our architecture scope, a host system is performed by a mono-processor CPU, as well as devices are performed

by up to 4 GPUs that each one contains about two or three hundred processor elements.

In summary, the two main benefits of GPUs are the following:

1. Computing Performance versus Price

Figure 0.1 shows the performance comparison between modern GPUs and Central Processing Units (CPUs). The ratio for peak floating-point (double precision [1]) calculation throughput is about 8 to 1. This ratio is merely the raw speed that the execution resources can potentially support in both GPU and CPU chips. Moreover, the price is a key aspect to consider. GPUs are graphics cards and their cost is about a few hundred dollars [2] and they are found even in notebooks. Some cases report an average of between 10x and 100x speed increase using General-Purpose computation on Graphics Processing Unit (GPGPU) for the same computations, depending on the algorithms and the hardware used for comparison [3].

1. Here, we emphasize rather double precision than single precision due to our main application domain. If we consider single precision calculation, the difference presented in the chart is fairly larger.

2. More expensive HPC cards based on GPU can reach a few thousand dollars.

3. http://developer.nvidia.com/cuda-action-research-apps

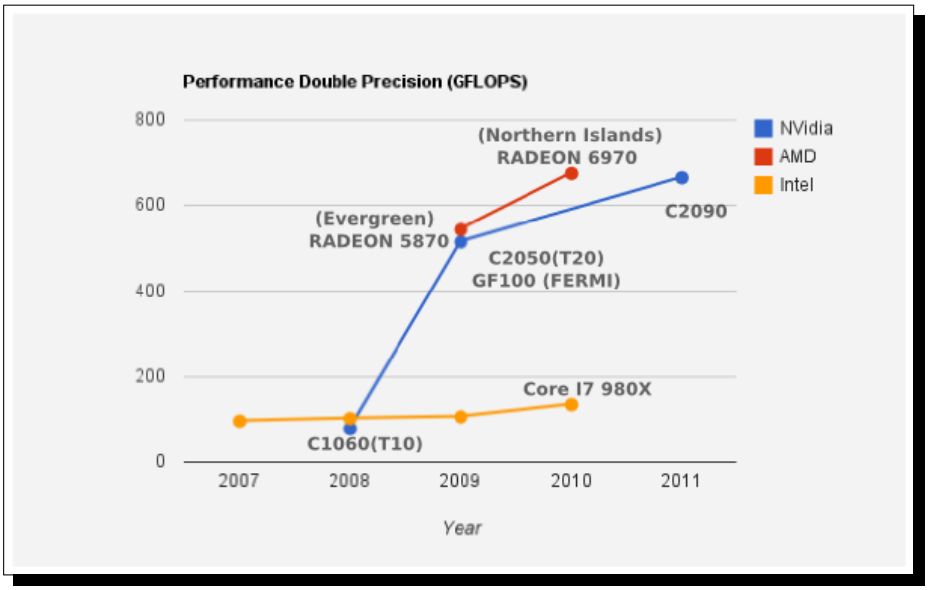


Figure 0.1: Performance gap between GPU and CPU

2. Green Computing versus Price

By their nature GPU processor cores can be aggregated readily into very dense multicore applications. There are GPUs with 1,000 cores in a single processor package with very low power consumption factor, maximizing performance per watt (optimizing overall costs). This factor is more important when we deal with clusters. Thus, GPGPU aims to use the environment efficiently and effectively. This is called *Green Computing* [108].

◦ *Issues on GPGPU*

According to [104] the adoption of GPGPU is driven by 7 issues. Some of these issues include:

- *Lack of developer expertise.*
  Even though GPGPU is based on existing programming languages, it requires a different model of programming concerning general purpose processors. Currently, developers are aiming to acquire necessary skills in order to adapt their programs to GPGPU.
- *C/C++ being the dominant starting point for GPGPU programming.*
  Vendor's API is usually based on C/C++; however, bindings for other languages have largely been disseminated.
- *Half of those considering GPGPU expect a 10x increase in performance.*
  Many available benchmarks show speed-up of about 10x. This is an attractive result though not always true for all applications. On the other hand, some benchmarks show better performance.
- *Debugging and designing parallel algorithms are the most difficult development task with GPGPU.* Available tools have constraints to interact with GPU processors. Debugging programs at runtime is hard to implement. Moreover, depending on the developing tool, it is not simple to integrate the vendor's profilers and programming environment.

High-Level Specification

There is a lack of high-level specification for GPGPU applications. However, the specification of parallel applications at higher levels is not recent. Indeed, in 1992, Jong et al. [32] proposed an approach that separates the algorithm specifications and the allocation of hardware resources to data and computations. In their approach, algorithms are formulated at an abstract level in a specification language having its own ideal virtual machine, thus preserving the parallelism inherent to the algorithm. Many others approaches appeared, and the high-level specification world has evolved. Today, there are several standards available to abstract specification of software depending on the application domain. Proprietary solutions as seen in Matlab Simulink® and Labview®, or standards as provided by Object Management Group (OMG) are alternatives to classical software specification based on hand written code. Therefore, the main issue, in this context, is to find the high-level specification solution that suits well to GPGPU developing. In summary, a solution that meets our objectives need to provide the means to specify an application, the expression of its parallelism, the platform architecture, and the link between logical and physical parts. Regarding the parallelism, an interesting aspect is conciseness. Thus, dozens as well as thousands of instances of a task must be expressed in a compact way with minimal complexity.

SCOPE AND CONTRIBUTIONS

Although parallel programming has a wide application range, we focus the work on simulation of electrical machines. More specifically, we focus on all operations that depend on linear algebra. Most of these operations can run in parallel and suit well to many-core architectures. Indeed, vector and matrix operations are data independent and do not require synchronization instructions. In addition, similar tasks as seen in intensive signal processing applications are also met by the same development methodology. In summary, our scope of work lies in introducing high-level specification into GPGPU development as seen in Figure 0.2.

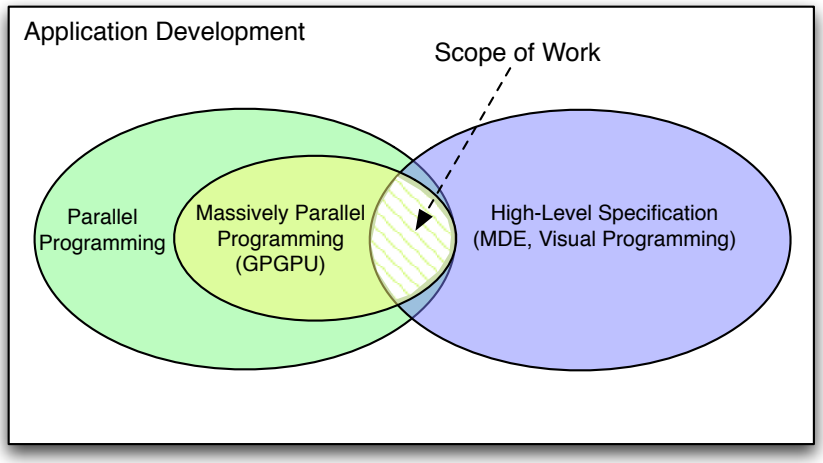


Figure 0.2: Scope of Work

Therefore, the contribution of this thesis addresses the following aspects:

An MDE Methodology for GPGPU Architectures

How to specify the models of application and platform for our target software and hardware?

We have chosen Model-Driven Engineering (MDE) as overall solution to our methodology because it aims to increase the benefit of the software development. In summary, this benefit is delivered in two basic ways:

1. *short-term productivity*: using rapid prototyping as part of a planned iteration in establishing application requirements;

2. *long-term productivity*: growing software organically, adding more function to systems or modifying them as they are run, used, and tested and according to running platform.

◦ *Modeling*

In the MDE context, we have found in the standard Unified Modeling Language (UML) and its profile Modeling and Analysis of Real-Time and Embedded Systems (MARTE) the elements that meet the earlier discussed concerns regarding high-level specification. MARTE is increasingly being used in academic and industrial tools to support hardware and software development. Among the features of UML and MARTE, the following ones are taken into account as reasons for our choice:

- *Clear separation between hardware and software specifications.* It enables the specialization of each component, providing a common way of modeling both hardware and software aspects of a whole system in order to improve communication between developers.
- *Existing tools and experienced developers for high-level modeling in UML.* It enables interoperability among development tools used for specification, design, verification, and code generation.
- *Parallel expressiveness available in MARTE.* With support to ArrayOL, it is able to express in compact form the potential parallelism of tasks and data and parallel hardware architectures, enabling factorization of repeated elements.

◦ *Code Generation*

Besides modeling, we define also the strategy to analyze a model specified with MARTE in order to produce a source code for the target platform. This process is not monolithic and is made by various layers, which take into account aspects previously designed.

◦ *Optimization*

We aim a methodology that produces an efficient code. In GPGPU, different programming practices, we are able to achieve higher performances. These practices include explicit cache handling, and attain speedup of about 10x[1] with respect to non-optimized codes. Thus, we integrate within the code generation engine some resources aiming at generating efficient code.

[1] *It is important to note that this is not always true. Usually, analysis by human obtains better results than automatic one.*

Metamodel for Data Allocation and Distribution

How to specify the data allocation
onto different memory address space?

We have to set a memory space and to define allocations for a host

and different devices without neglecting their intercommunication. Moreover, we have to gather all data elements defined in the application model that share the same memory address. To organize those aspects, we define a specific metamodel for memory handling.

Metamodel for Hybrid Applications

How to clearly separate CPU and GPU roles in the whole application?

We propose this metamodel as being the last one in our hierarchy of metamodels. It sums up all previous analyses aiming at creating a model with components that match elements in a real program. More precisely, it takes into account the hybrid structure (host and device) found in GPGPU architecture.

Model Transformations

How to transform higher level models to lower level models in order to attain the target code?

The core of our code generation approach lies mainly in the model transformations. We have defined several model transformations modules, which now, along with other ones, compose a complete chain of transformations for generating a functional and efficient code. These transformations analyse the model, and define or modify structures based on available metamodels that regard aspects such as data handling, scheduling, etc.

UML/MARTE-to-OpenCL Branch for Gaspard2

How to integrate the whole methodology into a single development environment?

Our methodology is inserted within a wider approach called Gaspard2. Indeed, we propose a new branch that allows transforming a model designed in UML/MARTE to a source code in OpenCL. The choice of OpenCL can be summarized in one reason: it is standard and independent of hardware vendor[2] .

[2] *CUDA is the programming model for NVidia's GPU. It is older and has a higher number of tools and case studies. However, it remains a proprietary solution and, at the moment, it cannot be used with other vendor's GPU.*

Increasing Performance on Heavy Simulation Applications

What benefits are observed in the whole application?

Performance is our main goal. Along with hardware price and a low curve learning methodology to develop applications, we contribute

to a solution to accelerate numerical algorithms on heavy problems. Indeed, we show that is not interesting to apply GPGPU on lightweight problems, and the turning point can be seen on benchmark results.

Fast Adaptation of Applications on Multiple Devices

How scalable is the code generation?

Those who already developed applications on GPGPU know that raising scale on devices implies complex changes on code. This is more noticeable in OpenCL programming model. We propose task allocations onto mono or multi-devices. Therefore, the complexity of the overhead to attain the multi-GPU feature is undertaken by the model compiling process.

Optimization Techniques taking advantage of MPP and MDE

How efficient is the generated application?

At model compiling time, we are able to take into account aspects that can be optimized. At first, knowing the GPGPU architecture allows us checking bottlenecks in performance and providing changes to overcome them. Second, enabling the integration between running results (profiling) and high-level models allows us tuning our models aiming at better results. Even if an optimized code depends mainly on the algorithm and the way in which the designer has specified it, we give resources to optimize the code to designers.

Application of the Methodology on Simulation of Electrical Machines

How much impact is achieved
on simulation of electrical machines?

The methodology hides details about programming GPGPU at low level. Therefore, we focus on designers that know well their algorithms and not necessarily the GPGPU programming model. Specifically for simulation of electrical machines, we provide a methodology that relies on replacing sequential implementation in existing simulation tools by parallel implementations. Also, replacing heavy computation modules by parallel solutions increases fairly performance and consequently decreases the time-to-market of the whole project.

Comparing to the OpenMP branch of Gaspard2

There is a previous work developed within Gaspard2 that also aims to provide a methodology to develop high performance applications in order to improve numerical simulation applications. This work involves an OpenMP transformation chain and is part of the Julien Taillard's thesis [137]. We present an overview of his work in Chapter 1. However, in order to emphasize some concerns that make our work stands out, we describe them as follow:

1. **Programming Model:** We focus on the OpenCL programming model. OpenCL has wider application and target architectures than OpenMP. Therefore, it brings new issues not addressed by OpenMP. The main issues are related to the explicit handling of data transfers and the grid of threads. However, Taillard did not address this concern. Furthermore, to a certain extent, OpenCL is capable to meet the application field of OpenMP, expanding it to other architectures that may integrate, for instance, hardware accelerators.
2. **High-level Specification:** Taillard's work proposes the usage of a non-standard UML profile to specify applications and architecture. Even though most concepts from this profile are now part of the standard MARTE profile for UML, MARTE provides a larger number of stereotypes related to embedded system domain. It includes the previous concepts required by the Taillard's approach and many other concepts aiming other applications. Moreover, MARTE is increasingly present in modeling tools, and its community and applications ensure future support and maintainability.
3. **Transformation Chain:** Despite the fact that Taillard's transformation chain is not defined by standard languages for specification of transformations, it is not structured in layers. Indeed, his transformations keep a monolithic structure and do not involve well defined functionalities (e. g. scheduling). In our approach, we were able to identify his contributions in common fields, such as task graph and task distribution. From these contributions, we redefined the structure of transformations in layers and added all common functionalities. Then, in order to meet the requirements of a GPGPU approach, we had to add other essential functionalities. These functionalities address specially data handling, performance, and the hybrid aspect of OpenCL applications that defines roles of host and device.

The concerns discussed above justify the need to address the problem in a different way. This is not simply due to differences between aimed execution platforms. This is above all due to choices and specific goals defined for implementation of the proposal. As a result,

we offer a better approach that takes into account its evolution and adaptability to future execution platforms.

OUTLINE

This dissertation is structured into four main portions.

Part I: State of the Art

The first part deals with related works to code generation, high-level specification for parallel programming, and basic concepts related to the framework used as a basis for developing our methodology.

- **Chapter 1: High-Level Modeling and Code Generation on HPC**
In this chapter, some important approaches in the state of the art on high-level modeling and code generation to High Performance Computing (HPC) systems are presented regarding the features relevant to our proposed approach.

- **Chapter 2: Gaspard2 as Code Generation Framework**
We have found in Gaspard2 framework important features which can benefit our proposal. Therefore, in this chapter, we present the design framework Gaspard2 as a solution to the development of high performance systems and which addresses MDE challenges.

Part II: Methodology Approach

Initially, in this part, we present the methodology itself. Moreover, the remaining chapters deal directly with the technical details related to code generation from a designed model.

- **Chapter 3: Creating Applications**
In this chapter, we seek to clarify our methodology to develop OpenCL applications from high-level models. Indeed, after presenting related works and Gaspard2 as framework for our approach, we analyze here two examples that use MARTE to specify them. Nevertheless, the examples are generic applications and aim only to describe how the methodology works.

- **Chapter 4: Metamodels and GPU**
In this chapter, we begin to explain technically the process behind the methodology. Basically, we point out principles of operation of GPUs, and then we present the main metamodels that statically provide a structure that supports these principles. We have proposed three novel metamodels. These metamodels focus on the different GPU features.

The chapter does not deal with all metamodels necessary to the whole code generation but only the proposed ones, and those related to GPU domain.

◦ **Chapter 5: Models towards Code**
In this chapter, we present several model transformations modules which now, along with other ones, are part of that we call Gaspard2 Model Transformation Library. As an MDE approach, the new branch proposed for Gaspard2 comprehends all models, metamodels, transformation modules, and, foremost, how to determine the compiling process layers in order to achieve all necessary model element analysis. In this chapter, we present our model transformation chain and how it works regarding the metamodels previously depicted in the earlier chapter as well as metamodels introduced in this chapter.

◦ **Chapter 6: Optimizations** Even if our proposal is based on abstract models in a very high-level programming, in this chapter, we present some key points used to optimize the generated code. These points are strongly linked to the running platform, i. e., CPU, GPU under OpenCL programming. We emphasize basically two strategies to try to optimize the code: memory issues and profiling integration.

Part III: Simulation of Electrical Systems

Our main goal is to allow creating high performance applications to simulate electrical systems. This part is composed of two chapters that present the simulation system CARMEL and a case study that replaces one of its modules (solver) with a GPGPU module.

◦ **Chapter 7: Electromagnetic Phenomenon and CARMEL**
This chapter present the scientific computation associated to the solution of electromagnetism problems that requires numerical methods in the discrete domain. So, in order to assure a better understanding of the decision taken in the case study, at first, we present part of the theory of the mathematical modeling of these problems in the continuous domain. Then, we introduce the Finite Element Method (FEM) as numerical method to solve these problems. Having this basic theory, we present the Code_CARMEL, a software that implements numerical methods for simulation of electrical machines.

◦ **Chapter 8: Conjugate Gradient Solver**
In this chapter, we present the direct application of the methodology to the parallelism of a solver in the context of Code_CARMEL. The objectives can be summarized into two main aspects: first, the high-level specification of the solver's algorithm using MARTE; second, the expected results by applying our solver module into the Code_CARMEL.

Part IV: Appendices

This thesis deals with a twofold research domain: computer science and electromagnetic phenomenon. For this reason, we added two appendices that have basic concepts of High Performance Computing (HPC) and Model-Driven Engineering (MDE). Those concepts are often referenced in the main text whenever their definitions are necessary.

◦ **Appendix A: High Performance Computing**
This appendix does not intend to be a complete study on HPC. However, the basic theory and specially the GPGPU and OpenCL worlds are presented here as a quick reference.

◦ **Appenidx B: Model-Driven Engineering**
Similar to Appendix A for HPC, this appendix presents basic concepts on MDE. More precisely, it deals with MDE philosophy and MARTE profile for UML. Again, those concepts are referenced in the main text and they enlighten some of MARTE packages.

Part I

STATE OF THE ART

1

HIGH-LEVEL MODELING AND CODE GENERATION ON HPC

Chapter Contents

Before presenting our approach, in this chapter, we discuss some important approaches in the state of the art on high-level modeling and code generation dedicated to HPC systems. There are many programming solutions available to create high performance applications. We emphasize here a twofold category for those solutions. On the one hand, we have languages that hide the low level from developers. In this case, developers specify their intentions at a high abstraction level and then a Domain-Specific Language (DSL) compiler is able to create automatically code for the desired platform. On the other hand, there are approaches that enable developers to use the power of languages for parallel applications based on templates or through directives simplifying the complex command syntax. In this chapter, we present some of those solutions, enlightening their advantages and drawbacks regarding the objectives seen in the introductory chapter.

1.1 HIGH-LEVEL SPECIFICATION APPROACHES

In this section, we emphasize 5 approaches based on high abstraction level specification. Some of them aim at creating high-performance applications, while others address modeling of systems.

1.1.1 *Gaspard2: OpenMP Branch*

Based on a previous version of Gaspard2, the OpenMP branch is a result of the Taillard's thesis [137]. His thesis' work focuses on parallel application development from its specification, maintenance, and code generation for the target platform. As our proposal, this work is result of a Model-Driven Engineering (MDE) methodology taking an application specification from a high-level abstraction UML model to a parallel code automatically generated by model transformations.

Our thesis is based on Model-Driven Engineering methodology. Important concepts of MDE are present in Appendix A.

One of our aims is extending the Taillard's work. However, we are obliged to take into account some limitations of his approach, as follow:

1. It does not use the standard Modeling and Analysis of Real-Time and Embedded Systems (MARTE) profile to specify the application.
2. Even if OpenMP is a parallel programming model, it has many differences from GPU's programming model, mainly the distributed memory aspect of CPU and GPU; OpenMP is oriented to shared memory and usually it is not suitable to thousands of threads management[3] .
3. The device multiplicity for OpenMP implies on a new available processor (or a few cores) to allocate new threads. In GPGPU, a new device implies on, for instance, 240 new cores with their own independent memory to allocate possible thousands of tasks instance (threads).
4. Even if every produced application can be always optimized, the compiler within model transformations takes into account just the model as it was designed (even if a memory management is proposed), the model designer is responsible for possible necessary changes in the model in order to attain a better performance level.
5. Except for the thread grid definition, the management of threads dispatching is entirely made by the GPU controller. Differently, Taillard makes use of polyhedron models to optimize loops and distribute sub-tasks in OpenMP.

[3] *At least in current platforms in which OpenMP is used.*

However, the OpenMP branch is the source of inspiration for many features available in our GPU approach. Indeed, the parallel expression based on ArrayOL concepts are found in OpenMP branch and available on Gaspard2, as well as the conditional loop exit control

necessary in applications such as Conjugate Gradient (CG) explained in Chapter 8.

1.1.2 *Archi-MDE*

In [98], Lugato et al. propose UML-HPC, or *UML Profile for High Performance Computing* as a set of profiles for UML that allows designing models for structures, data types, and data processing. From the user's viewpoint, UML-HPC performs the design of a scientific computing software package with the help of the following concepts: modules (HPCModule), and methods (HPCMethod). These concepts are deliberately closer to those currently envisioned by designers with Fortran language. Archi-MDE is an environment (*cf.* Figure 1.1) that proposes the integration of UML-HPC as part of a complex system that provides several services such as model-checking and static profiling.

However, we are not able to verify any case study providing neither functional nor performance results. We suppose that this approach aims at creating applications in high performance Fortran, i. e. OpenMP Fortran, nevertheless it is not clear what is the model of computation or how threads can communicate in a multi-threaded environment. As a non-standardized approach, no extension and no compatibility are provided with current standards to design HPC applications.

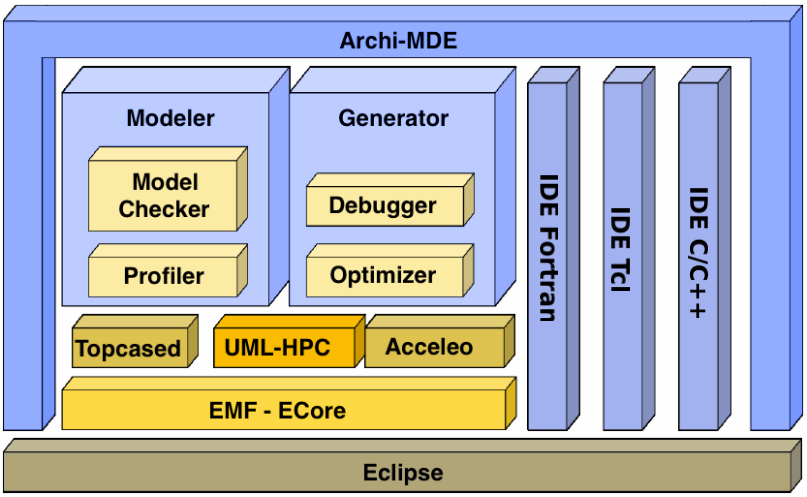


Figure 1.1: Architecture of the plug-in Archi-MDE. *Source: author's article* [98].

1.1.3 *Simulink*

Simulink is a dynamic simulation environment that is part of the Matlab package marketed by the Mathworks Inc [99]. Matlab was originally written by Cleve Moller in order to aid users in using two

Fortran subroutine libraries designed to solve linear and eigenvalue problems. In 1984 Matlab was re-written in C; it contains powerful visualization, differential equation solving, and matrix handling functionality. Additional functionality can be added to Matlab by using additional subroutines called Toolboxes. Though called a Matlab Toolbox, Simulink is now an essential component of the Matlab/Simulink suite. Simulink can be used to model and simulate dynamic systems that are represented by differential equations, algebraic relations, and finite state machines, using a library of standard components (called blocks). Simulink has a large library of blocks that can be used to model even very complex non-linear systems. As part of the standard library, Simulink includes a block intended to allow users to develop and implement their own custom routines: the S-Function block. Though originally intended as a dynamic systems simulator, Simulink's functionality has been expanded; an example of a state-of-the-art use of Simulink is in the controlled rapid prototyping process. However, there is a lack of resources to model the parallelism of tasks and data, and resources to specify the running architecture. A Simulink model is represented graphically by means of a number of interconnected blocks. Lines between blocks connect block outputs to block inputs. Blocks may have states, which may consist of a discrete-time and a continuous-time part. In [80], Hooman et al. present an approach that couples UML and Simulink. Nevertheless, some issues related to time modeling impose constraints to the generality of the approach. Figure 1.2 presents a typical modeling environment of Simulink.

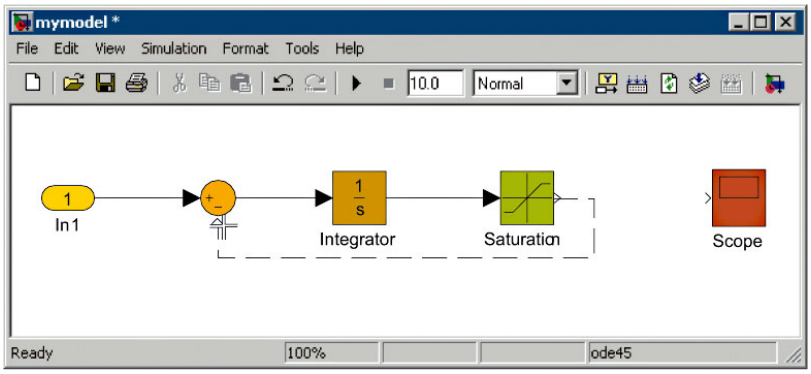


Figure 1.2: Designing Models on Simulink. *Courtesy: MathWorks*™

1.1.4 *OpenModelica*

OpenModelica is an open source implementation of a Modelica compiler, simulator and development environment for research as well as for educational and industrial purposes. OpenModelica is developed and supported by an international effort, the Open Source

Modelica Consortium (OSMC) [119]. OpenModelica consists of a Modelica compiler, OMC, as well as other tools that form an environment for creating and simulating Modelica models (*cf.* Figure 1.3).

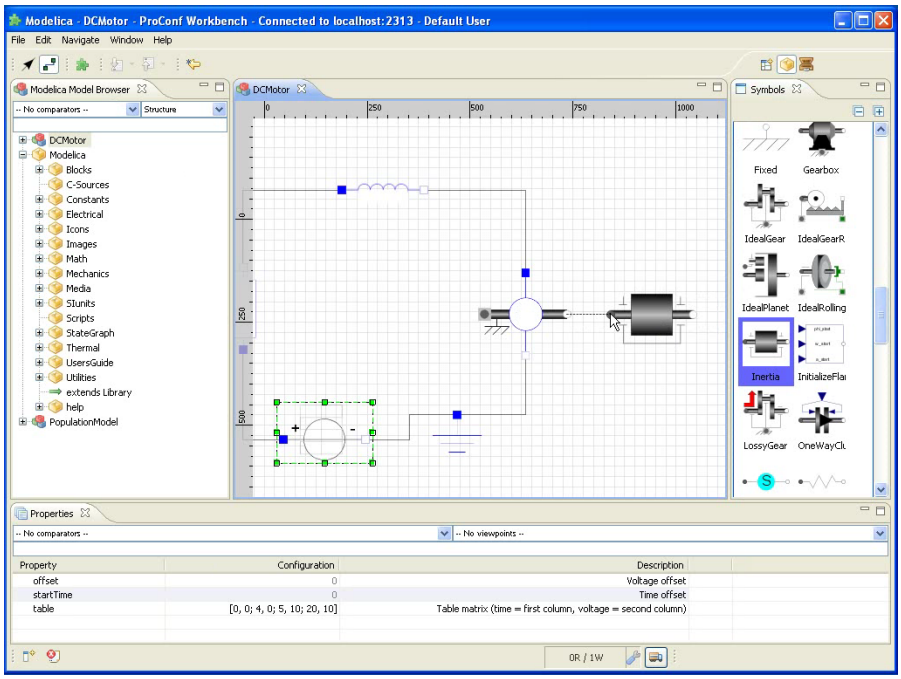


Figure 1.3: Modeling a DC Motor in OpenModelica IDE

OpenModelica provides a graphic editor similar to Simulink. The editor allows us to model a system using blocks of elements available in its libraries. These blocks can represent power sources, adders, transfer functions, etc. Libraries are divided in domain groups.

Regarding the compilation process, it differs quite a bit from programming languages such as C, C++ and Java. The OpenModelica front-end will first instantiate the model, which includes among other things the removal of all object-oriented structure, and type checking of all equations, statements, and expressions. The output from the OpenModelica front-end is an internal data structure with separate lists for variables, equations, functions and algorithm sections. For-equations are currently expanded into separate equations. This means that currently each is analyzed independently. This is inefficient for large arrays. From the internal data structure executable simulation code is generated. The mapping of time-invariant parts (algorithms and functions) into executable code is performed in a relatively straightforward manner: Modelica assignments and functions are mapped into assignments and functions respectively in the target language of C++. For in-depth details about the compilation process of OpenModelica refer to [123].

1.1.5 *Syntony*

Dietrich et al. [46] propose an improved model-driven software development process for numerical algorithms. Their approach is a twofold process: first, the high-level system behavior is modeled using UML class and activity diagrams. Then, the modules which should be copied from the existing code base are specified in the UML model. Typically, all hardware-specific and computationally intensive modules will be included. The tool Syntony is able to automatically generate a full working code from these UML models.

In detail, the process consists of several steps:

1. implementation of basic data types, low level algorithms, and I/O routines efficiently in C++ or even on specific hardware like GPUs;
2. design of classes in UML or auto-generate class model from existing framework;
3. design of high level algorithms in UML activity diagrams;
4. translation of UML model into C++ code (Syntony);
5. Compilation and execution of the code on a specific platform or from the GUI.

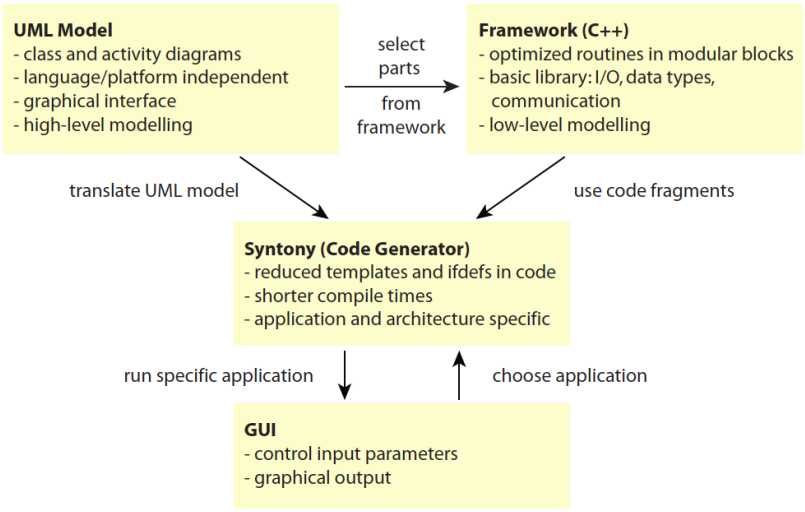


Figure 1.4: Overview of Syntony approach. *Source: author's article* [46].

Figure 1.4 illustrates the whole process. In fact, the application designer choose pre-defined elements such as routines, datatypes, communication modules, referred to as Framework(C++). Syntony generates code regarding the target architecture based on templates. A GUI allows us to choose running specific application previously generated in Syntony.

It is important to observe that these processes are not suitable for generic applications. Instead, it is meant to be used for a big collec-

tion of similar applications like present in the discussed framework. Syntony is intended to be extended to generate code for different hardware platforms, for example GPUs, even though no strategy to achieve this goal is explicit.

1.2 EXTENSIONS FOR PROGRAMMING LANGUAGES

Although GPU programming is easier than before when we were obliged to use languages for graphics operations, the new approaches such as CUDA and OpenCL remain difficult for neophytes. Many solutions based on directives, libraries for several languages, pragmas, and others have been emerging to facilitate the development. These sections intend to present some of the main approaches that use these strategies.

1.2.1 *Mint Programming Model*

Mint is a programming model that enables the non-experts to enjoy the performance benefits of hand coded CUDA without becoming entangled in coding details. Mint programming model has been implemented as a source-to-source translator that generates optimized CUDA C from traditional C source. The translator relies on annotations to guide translation at a high level. The set of pragmas is small, and the model is compact and simple. Yet, Mint is able to deliver performance competitive with hand-optimized CUDA. In the paper [143], Unat et al. show that, for a set of widely used stencil kernels, Mint realized 80% of the performance obtained from aggressively optimized CUDA on the 200 series NVIDIA GPUs. Their optimizations target three dimensional kernels, which present a daunting array of optimizations.

Compute Unified Device Architecture is a parallel computing architecture developed by Nvidia. Even though CUDA is platform-dependent, many solutions use CUDA as a target language due to its large number of developers. CUDA works like OpenCL. It is slightly introduced in Appendix A.

1.2.1.1 *C to CUDA Translation*

To construct the source-to-source translation and analysis tools, the ROSE compiler framework [94, 127] is used. ROSE is an open source software developed and maintained at Lawrence Livermore National Laboratory. ROSE is a convenient tool for developing this kind of infrastructure because it provides an API for generating and manipulating memory representations of Abstract Syntax Trees (ASTs).

Figure 1.5 shows the modular design of the Mint translator and the translation workflow. The input to the compiler is C source code annotated with Mint pragmas. The *Pragma Handler* parses the Mint directives and clauses. Once the translator has constructed the AST, it queries the parallel regions containing data parallel for-loops. Directives in a candidate parallel region go through several

transformation steps inside the Baseline Translator: *Outliner, Kernel Conguration, Argument Handler, Memory Manager,* and *Thread Scheduler*.

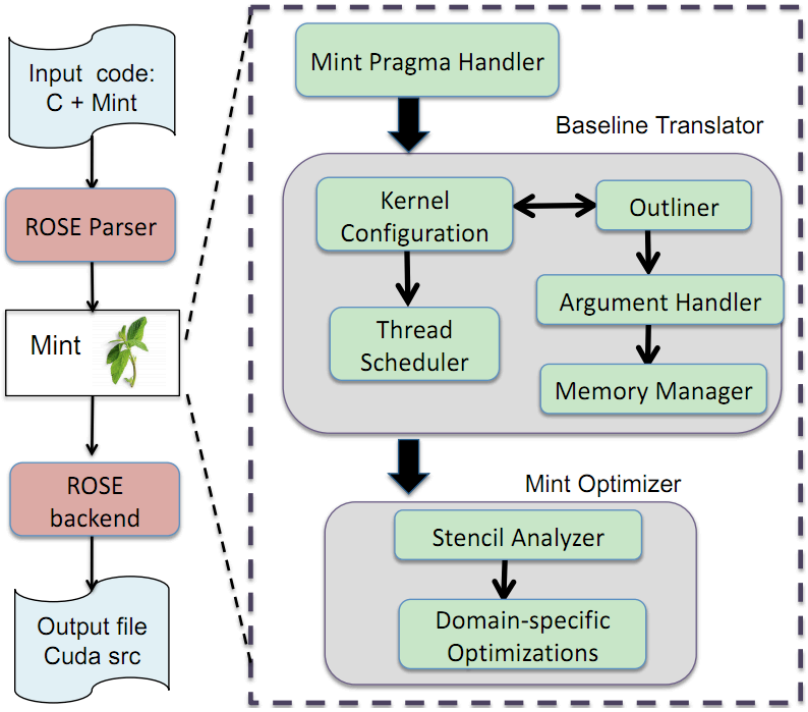


Figure 1.5: Modular design of Mint Translator and translation flow. *Source: author's article* [143]

An obvious limitation for this translator is that it is domain-specific. Mint targets stencil computations and its optimizations are specific to this problem domain. Although it is not a generic approach, Mint can incorporate domain-specific optimizations into the compiler, resulting in improved performance.

1.2.2 *OpenHMPP*

Based on a set of directives, OpenHMPP Standard [120] is a programming model designed to handle hardware accelerators without the complexity associated with GPU programming. This approach based on directives has been implemented because they enable a loose relationship between an application code and the use of a hardware accelerator. The OpenHMPP directive-based programming model offers a syntax to efficiently offload computations on hardware accelerators and optimizes data movement to/from the hardware memory. The model is based on works initialized by CAPS (Compiler and Architecture for Embedded and Superscalar Processors) [22], a common project from INRIA, CNRS, the University of Rennes 1 and the INSA of Rennes.

OpenHMPP is based on the concept of codelets, functions that can be remotely executed on Hardware Accelerators. A codelet has the following properties:

- It is a pure function:
  - it does not contain static or volatile variable declarations nor refer to any global variables except if they have been declared by a HMPP directive "resident";
  - it does not contain any function calls with an invisible body (that cannot be inlined). This includes the use of libraries and system functions such as malloc, printf, etc.;
  - every function call must refer to a static pure function (no function pointers).
- It does not return any value (void function in C or a subroutine in FORTRAN).
- The number of arguments should be fixed (i.e. no variable number of arguments like *vararg* in C).
- It is not recursive.
- Its parameters are assumed to be non-aliased.
- It does not contain callsite directives (i. e. RPC to another codelet) or other HMPP directives.

Listing 1.1: OpenHMPP example 1

```
1  /* declaration of the codelet */
2  #pragma hmpp simple1 codelet, args[outv].io=inout, target=CUDA
3  static void matvec(int sn, int sm, float inv[sm], float inm[sn][sm],
       float *outv){
4      int i, j;
5  for (i = 0 ; i < sm ; i++) {
6        float temp = outv[i];
7  for (j = 0 ; j < sn ; j++) {
8          temp += inv[j] * inm[i][ j];
9      }
10    outv[i] = temp;
11 }
12 int main(int argc, char **argv) {
13    int n;
14 ........
15 /* codelet use */
16 #pragma hmpp simple1 callsite, args[outv].size={n}
17 matvec(n, m, myinc, inm, myoutv);
18    ........
19 }
```

In Listing 1.1, we can observe the codelet declaration (lines 1-11) having CUDA language as target. In lines 16,17 the codelet call. The code is a conventional C code, however *pragmas* are inserted just before function call and declaration in order to ensure the settings intended by developers.

OpenHMPP is a powerful solution for manycore platforms and it does not imply big changes in previous existing codes. However, it

requires low level understanding about architecture details in order to attain better performances. For instance, in Listing 1.2 illustrates a situation in which we have to allocate data before running the loop. The memory allocation and upload of the input data are done only once outside the loop and not in each iteration of the loop. The synchronize directive (line 8) allows to wait for the asynchronous execution of the codelet to complete before launching another iteration. Finally the *delegatedstore* directive (line 10) outside the loop uploads the *sgemm* result. Regarding languages such as OpenCL, these operations remain complexes, even though they are simpler when we just need to modify already existing C codes.

Listing 1.2: OpenHMPP example 2

```
1  int main(int argc, char **argv) {
2  #pragma hmpp sgemm allocate, args[vin1;vin2;vout].siez={size,size}
3  #pragma hmpp sgemm advancedload, args[vin1;vin2;vout], args[m,n,k,
       alpha,beta]
4
5  for ( j = 0 ; j < 2 ; j ++) {
6  #pragma hmpp sgemm callsite, asynchronous, args[vin1;vin2;vout].
       advancedload=true, args[m,n,k,alpha,beta].advancedload=true
7   sgemm (size, size, size, alpha, vin1, vin2, beta, vout);
8  #pragma hmpp sgemm  synchronize
9  }
10 #pragma hmpp sgemm delegatedstore, args[vout]
11 #pragma hmpp sgemm release
```

1.2.3 *Java OpenCL Bindings*

Some developers prefer Java [65] as a programming language. Java derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities. There are a few libraries providing Java bindings for Open Computing Language (OpenCL). The goal of these libraries is to provide an object-oriented abstraction of OpenCL for Java. This simplifies the usage and may be considered more convenient for most Java programmers. The functions are provided as static methods, and semantics. Signatures of these methods have been kept consistent with the original library functions, except for the language-specific limitations of Java. The OpenCL API may be very verbose at some points, and this is not hidden or simplified, but is simply offered by the library as-it-is.

For instance, Listing 1.3 shows the process to obtain a device ID in JOCL [84]. There is almost no difference in the same process in C language. However, this code is part of a Java code and, thus, it can be part of a large application entirely developed in Java.

Listing 1.3: JOCL example

```
1  // Obtain a device ID
```

```
2 cl_device_id devices[] = new cl_device_id[numDevices];
3 clGetDeviceIDs(platform, deviceType, numDevices, devices, null);
4 cl_device_id device = devices[deviceIndex];
```

1.2.4 *Matlab and Matlab-like on GPU*

In recent years, several approaches have been enabling Matlab code [100, 78, 66, 2] and open-source alternatives such as Octave [50, 139] to run on the GPU. Matlab™ GPU support is available in Parallel Computing Toolbox™. Using Matlab for GPU computing allows us to take advantage of GPUs without low-level C or Fortran programming. Matlab supports Nvidia CUDA-enabled GPUs with compute capability version 1.3 or higher, such as Tesla 10-series and 20-series GPUs. It provides the base for GPU-accelerated Matlab operations and allows for integrating the existing CUDA kernels into Matlab applications.

The example provided in Listing 1.4 illustrates the usage of the power of GPU-enabled Matlab. In the line 1, we create a matrix 1000-by-1000 of single precision elements directly in GPU memory calling the `gpuArray` special function and it is referenced as `Ga` variable. Then, we do the Fast Fourier Transform (FFT) on this variable in GPU and the output is the variable `Gfft` also in GPU memory (as seen in lines 2,11). The following operation in the line 3 is executed by the GPU. Finally, we gather the result to CPU memory in variable G[4] as seen in lines 4,8.

[4] *Single Precision=4 bytes. 1000x1000 elements=1,000,000 elements. 4x1million=4million bytes.*

Listing 1.4: Matlab GPU example

```
1 Ga = gpuArray(rand(1000, 'single'));
2 Gfft = fft(Ga);
3 Gb = (real(Gfft) + Ga) * 6;
4 G = gather(Gb);
5 whos
6  Name       Size         Bytes  Class
7
8  G       1000x1000     4000000  single
9  Ga      1000x1000         108  parallel.gpu.GPUArray
10 Gb      1000x1000         108  parallel.gpu.GPUArray
11 Gfft    1000x1000         108  parallel.gpu.GPUArray
```

This solution is very interesting for Matlab programmers. However, it remains dedicated to specific available functions where the parallelism is evident and, for the time being, it does not allow further configurations such as multi-GPU.

1.2.5 *PyOpenCL*

Interpreted languages such as Python [125] are traditionally used in IO-bound applications where their lack of high CPU-bound performance does not matter. Moreover, Python is a high-level programming

language and object-oriented. This means lower learning curve. To add more complex operations, however, extensions written in C (or C++) can be used to implement these computationally heavy operations that can be called from Python as if they were native functions.

The project PyOpenCL purposes adding OpenCL support to Python. In [92], Pinto et al. introduce PyCUDA and PyOpenCL as a combination of a dynamic, high-level scripting language with the massive performance of a GPU as a compelling two-tiered computing platform, potentially offering significant performance and productivity advantages over conventional single-tier, static systems. The concept of GPU run-time code generation (RTCG) presented in their article is simple and easily implemented using existing, robust infrastructure. Nonetheless it is powerful enough to support the creation of custom application-specific tools by its users. Analyzing the code in Listing 1.5, we can see how this approach allows us to interact with OpenCL programming. Although, originally, Python language hides the low-level details about memory management, in this example, in lines 5,6 we declare explicitly two variables *float32(double)* with 50000 random elements, then in lines 12,13 we allocate and transfer these variables contents to the device. The OpenCL code is provided and built as-it-is in lines 16-23, and then executed (line 25). The function `cl.enqueue_copy` (line 28) transfers the result back to host.

Listing 1.5: PyOpenCL example

```
1  import pyopencl as cl
2  import numpy
3  import numpy.linalg as la
4
5  a = numpy.random.rand(50000).astype(numpy.float32)
6  b = numpy.random.rand(50000).astype(numpy.float32)
7
8  ctx = cl.create_some_context()
9  queue = cl.CommandQueue(ctx)
10
11 mf = cl.mem_flags
12 a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
13 b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
14 dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)
15
16 prg = cl.Program(ctx, """
17     __kernel void sum(__global const float *a,
18     __global const float *b, __global float *c)
19     {
20       int gid = get_global_id(0);
21       c[gid] = a[gid] + b[gid];
22     }
23     """).build()
24
25 prg.sum(queue, a.shape, None, a_buf, b_buf, dest_buf)
26
27 a_plus_b = numpy.empty_like(a)
28 cl.enqueue_copy(queue, a_plus_b, dest_buf)
```

```
29
30 print la.norm(a_plus_b - (a+b))
```

Similarly to Java bindings, PyOpenCL offers resources to run OpenCL code directly in a Python code. This is interesting for two reasons: first, we can use GPUs taking advantage of a script language; second, larger applications developed in Python can be extended to run GPU applications. However, the approach remains only a binding between OpenCL[5] applications and Python.

[5] *Indeed, there is no change in OpenCL code.*

1.2.6 *SAC*

Single Assignment C (SAC) is a full, standalone functional and data-parallel programming language. Most of its basic language constructs are identical to those of C, not only with respect to their syntax but also with respect to their semantics. Despite this rather imperative look and feel, a side-effect free semantics is enforced by the exclusion of a few features of C, most notably the notion of pointers. As a replacement, the language incorporates extensive support for compiler-managed multi-dimensional arrays.

The language features a special construct, WITH-loop, for expressing data-parallel operations. A complete discussion can be found in [134]. A WITH-loop expression in SAC consists of one or more generator parts and an operation. The latter determines the overall behavior of a WITH-loop. Various application studies have demonstrated that the compiler generated codes can achieve: (i) competitive sequential runtimes comparable to those of hand-written C and FORTRAN codes, and (ii) almost linear speedups from auto-parallelization for shared memory systems.

The GPU backend of SAC allows to generate CUDA code from a high-level functional array programming language. The compiler targets the data parallel loops, WITH-loops in SAC for parallel execution. In addition to mapping the WITH-loops to CUDA kernels, it performs additional transformations for improving the performance even further. Their findings allow minimizing redundant data transfers as a key optimization technique.

Listing 1.6: Example CUDA-WITH-loop with data transfers inserted.

```
1 Adev = host2device(A);
2 Bdev = cuda_with {  //cuda_with are eligible WITH-loops for GPGPU
3       ( [1,1] <= iv=[i,j] < [4095,4095]) {
4         res = 0.25*(Adev[i][j-1] + Adev[i][j+1] + Adev[i-1][j] + Adev
              [i+1][j]);
5       }:res;
6    }:modarray(Adev);
7 B = device2host(Bdev);
```

Listing 1.7: Translating example CUDA-WITH-loop to kernel function.

```
cudaMemcpy(A, Adev, size(A), cudaMemcpyHostToDevice);
int d0 = 4095 - 1;
int d1 = 4095 - 1;
dim3 grid( d1/BLOCKSZ+1, d0/BLOCKSZ+1);
dim3 block( BLOCKSZ,BLOCKSZ);
CUDA kernel<<<grid,block>>>(Bdev,4096,4096,1,1,4095,4095,Adev);
cudaMemcpy(B, Bdev, size(B), cudaMemcpyDeviceToHost);

__global__ void CUDA_kernel (float Bdev, int shp0, int shp1, int lb0,
      int lb1, int ub0,int ub1,float Adev)
{
  int i=blockIdx.y*blockDim.y+threadIdx.y+lb0;
  if( i >= ub0) return;
  int j=blockIdx.x*blockDim.x+threadIdx.x+lb1;
  if( j >= ub1) return;
  int wlidx = i*shp1+j;
  res = 0.25*(Adev[i][j-1] + Adev[i][j+1] + Adev[i-1][j] + Adev[i+1][
      j]);
  Bdev[wlidx] = res;
}
```

1.3 OTHER CONTRIBUTIONS

The following approaches are considered important but they were not analyzed in more details.

Sadayappan et al. [132, 107] propose to develop a programming environment for easing the development of portable high-performance applications for GPUs and accelerators, by automatic generation of OpenCL code from annotated C programs provided by the user. The proposed work is motivated by recent advances in polyhedral based approaches for powerful transformations of affine computations that have enabled the development of the Pluto automatic parallelization/optimization system. The developments will result in enhancement of the widely used gcc compiler providing broad impact on sciences. The education of students engaged in the work is essential to the project. Moreover, they propose techniques for reducing the number of unroll factors evaluated, based on the characteristics of the program being compiled and the device being compiled to. They use these techniques to evaluate the effect of loop unrolling on a range of GPGPU programs and show how to correctly identify the optimal unroll factors.

The PGI Accelerator compilers [140] are commercial tools that automatically analyze whole program structure and data, split portions of the application between the CPU and GPU as specified by user directives, and define and generate an optimized mapping of loops to automatically use the parallel cores, hardware threading capabilities and SIMD vector capabilities of modern GPUs. In addition to OpenMP-like directives and pragmas that specify regions of code or

functions to be accelerated, the PGI Accelerator compilers support user directives that give the programmer fine-grained control over the mapping of loops, allocation of memory, and optimization for the GPU memory hierarchy.

1.4 COMPARATIVE TABLE OF FEATURES

Table 1.1 summarizes the existing approaches previously presented. Here, we enlighten the main concerns regarding our proposal subject of this thesis. One of the main motivations for starting a new approach lies in the use of profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE). MARTE is the standard that is becoming adopted by many industrial and academics for real-time and embedded systems specification. Further, in its Repetitive Structure Modeling (RSM) package, MARTE allows us specifying the data and task parallelism of applications compactly. In the table, first line describes features that we consider essential in an approach of code generation to GPGPU. Obviously, the approaches presented in this chapter have other undeniable features and a development background that any new proposal will probably lack. However, we aim to emphasize the difficulty in gathering ideal solutions in one single approach.

1.5 CONCLUSION

In summary, in this chapter, we enumerate some of the approaches related to Model-Driven Engineering (MDE) and General-Purpose computation on Graphics Processing Unit (GPGPU) available in High Performance Computing (HPC) literature. Some of them enable high level specifications (e. g. UML diagrams, visual programming) of applications and others do code generation for GPGPU from existing languages. Unlike our approach, they do not offer a full integration between high-level abstraction models and code generation. Afterwards, optimization techniques based on model analysis and integration between run-time profilers and those models are key points aimed by our approach in order to provide better applications. Furthermore, one of the important concerns, proposed in our approach and not found in others, is the ability of modeling details about the hardware platform such as the device multiplicity.

Table 1.1: Approaches Comparison

Approach	Specification of Application	Architecture Definitions	Parallelism Support	GPGPU	Optimizations	Profiling Integration
Desired Features	UML profile for MARTE	Defining hardware details such as memory hierarchy, host and device separation, and device multiplicity taking advantage of MARTE	Yes. Specified on higher-levels	Yes	By memory analysis and pulling up profiling results	Yes. Integrating profiling and hardware information in order to provide an aided design
Gaspard2 OpenMP	UML former profile for Gaspard2	Yes	Yes. Specified compactly on higher-levels	No	Some level on memory use	No
Archi-MDE	UML and HPC non-standard profile	No	Yes, but there is no information about how to specify it	No	It depends on the application specification	No
Simulink	Toolboxes(graphical blocks) and script bindings	No	Yes, without specification by the user	Via Parallel Computing Toolbox™	At function's algorithm level	No
OpenModelica	Graphical blocks (similar to Simulink)	No	Yes, without specification by the user	No fully integrated but there are some works using this feature	No	No
Syntony	UML models for pre-defined function modules	No	No	Not yet	At function's algorithm level	No
Mint	Pragmas for stencil applications	No	Yes, using code annotations on loop-nests	Yes	Already optimized for domain-specific	No
OpenHMPP	Pragmas for ordinary functions in C or Fortran (codelets)	No	Yes, using code annotations	Yes	At memory use levels	No
Java OpenCL	Java	No	Yes, at same OpenCL level	Yes	No	No
MATLAB	Command line and scripts MATLAB	No	Yes, without specification by the user	Via Parallel Computing Toolbox™	At function's algorithm level	No
PyOpenCL	Scripts Python	No	Yes, at same OpenCL level	Yes	No	No
SAC	New language for static assignment and using C-like syntax	No	Yes	Yes	Yes. It does it at several compilation levels	No

2

GASPARD2 AS CODE GENERATION FRAMEWORK

Chapter Contents

For reasons already explicit in the introduction chapter, we have chosen MARTE as a profile to refine our UML models and to express our needs when specifying a Massively Parallel Processing (MPP) application. In this field, we searched for a framework approach which could address the following aspects:

- **Modeling**: using MARTE for software and hardware specifications;
- **Model Transformation**: providing a framework to chain transformations in order to refine models and to generate code;
- **Traceability**: offering support to keep the link between the generated code and the high level model;
- **Modular**: assuring a non monolithic model compiler allowing to add extra features that we find it necessary.

We have found all these features in Graphical Array Specification for Parallel and Distribute Computing (GASPARD2) and we have decided to implement our approach as a new branch in the target platforms tree of GASPARD2. Therefore, in this chapter, we present the design framework GASPARD2 [59] as a solution to the development of high performance embedded systems and which addresses the following challenges:

Gaspard2 has been developed by DaRT team of the INRIA LILLE-NORD EUROPE and the LIFL (Laboratoire d'Informatique Fondamentalle de Lille).

- **Parallelism:** although there are parallel approaches in programming environments (*cf.* Appendix A), hand-coding is still widely adopted in the development of applications. Parallelism is handled at a low level by different expert teams which have a deep knowledge of the system (both hardware and software parts) in order to attain the performance requirements. Hand-coding is clearly not suitable for an efficient development of large embedded systems because it is very tedious, error-prone and expensive. In order to minimize the complexity of parallel programming, it is necessary to achieve a development degree where the design of high performance systems needs a new expressive parallel programming paradigms. Such paradigms have to provide some efficient way to separately represent, at a higher level, all the potential parallelism that is inherent to both software applications and hardware architectures. Furthermore, they must be rich enough to explicitly express different mapping possibilities of the application on the architecture, taking into account the parallelism of the whole application. Finally, to increase performance, the intermediate layers between the application level and the hardware architecture should be removed as much as possible.
- **Abstract Models:** Nowadays, the design of most systems is facing a strong pressure to decrease the time-to-market while the complexity of these systems increases. System developers have to rely on some costless means allowing them to simulate and analyze the behavior of designed systems before their realization. Design abstraction offers a possible solution to address the above issue concerning the time-to-market and complexity dilemma, and the systems development cost. More concretely, one needs models that capture the strict relevant information depending on the required abstraction level. The global complexity of a system is addressed from multiple viewpoints or abstraction levels, so that one is able to easily focus on some specific aspects. Abstract models grant an efficient design reuse, typically through incremental refinements from higher level models to lower level models. Here, by refinement, we mean a transformation that makes a given model more concrete with regard to a target representation (which is in general more precise than its current representation). On the other hand, since models are often executable and verifiable, they also serve as an interesting support

for both behavioral simulation and property analysis without having necessarily the actual implementation of systems. In some cases, they are even used to automatically synthesize this implementation. Finally, abstract models enable to deal with the heterogeneity of a system since its components can be handled at high description levels that abstract away the specific details of each component.

- **Co-Design:** the development of a complete solution usually starts with the concurrent design, or co-design, of both software application and hardware architecture. Then, the application part is mapped onto the hardware part, during the allocation phase. Therefore, simulation models from several abstraction levels are generated for the whole system. The different aspects of this development process are potentially handled by different domain experts who must communicate safely in order to achieve the expected design. In such a context, the design and analysis activities become very difficult due to the ever increasing complexity of modern systems. As a consequence, the productivity of designers falls down. Another critical aspect concerns the performance improving based on changes in models, i. e., how to re-design the system aiming better results.

2.1 INTRODUCTION TO THE FRAMEWORK

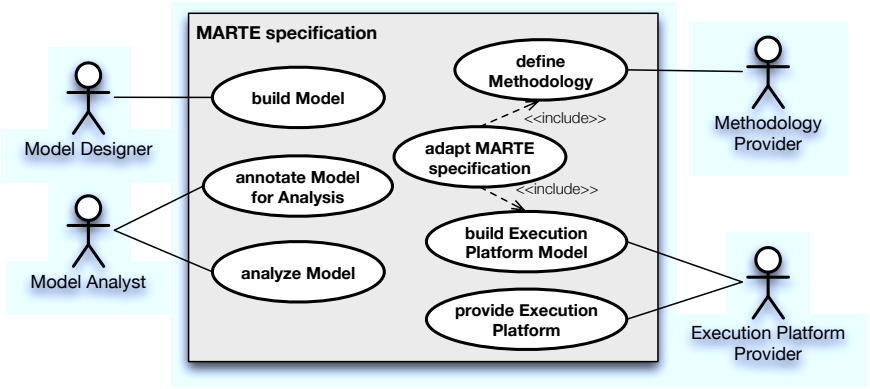


Figure 2.1: MARTE Use Case

Before presenting the framework, we will define the actor's roles according to MARTE. Figure B.2 presents the use case diagram that defines those roles. Based on this diagram and adopted as terminology for this thesis, we call *model designer* the user responsible for creating the application's model. Conversely, GASPARD2 has the role of *methodology provider* and once the execution environment is taken into account by GASPARD2, it can be also considered the *execution platform*

A more detailed introduction to MARTE profile is presented in Appendix B. There is a section which shows MARTE and its packages' descriptions and relationships.

provider. The *model analyst* is not mandatory to us. However, the most times when we have to analyze profiling results, the *model designer* becomes the *model analyst*.

In GASPARD2 context, we call framework an environment that provides to designers, at least, the following means:

- a formalism for the description of a whole application and platform at a high abstraction level,
- a methodology covering all system design steps,
- and a toolset that supports the entire design activity.

The design of SoCs in Gaspard specifically relies on the repetitive Model of Computation (MoC) [20], which offers a very suitable way to express and manage the potential parallelism in a system. This MoC is inspired by ARRAYOL(*cf.* Subsection B.2.2), a domain-specific language originally dedicated to intensive signal processing applications. It extends the basic notions of this language and offers an elegant and very expressive way to describe both task parallelism and data parallelism, and the combination of both.

The repetitive MoC is used in GASPARD2, via the MARTE standard profile and more precisely its Repetitive Structure Modeling (RSM) package(*cf.* Subsection A.4.2.5), to describe the parallel computations in the application software part, the parallel structure of its hardware architecture part, and the association of both parts. The resulting abstract models are afterwards deployed towards specific implementations. Finally, different automatic refinements from the higher abstraction level are defined, according to MDE paradigm, towards lower levels for various purposes: simulation at different abstraction levels with SystemC [13], hardware synthesis with VHDL [96], formal validation with synchronous languages [58], high performance computing with OpenMP Fortran and C [138]. MDE enables to clearly identify different intermediate abstraction levels. Thus, it facilitates the decomposition of the refinement process into successive steps.

2.1.1 *Gaspard2 Extensions*

One of the most important features in GASPARD2 is its ability to extend UML to bridge a gap of concepts not provided by MARTE. The extensions proposed are available by two ways: first, adding the deployment concerns by a UML profile (*cf.* Section 2.4) along with MARTE; second, offering an extensible library of metamodels as extension of the MARTE metamodel.[6]

In general, those extensions can be divided into two categories as illustrated in Figure 2.2:

1. **Core Extensions**: they are all metamodels related to common concerns, e. g. scheduling, tiler processing.

[6] *Originally MARTE is implemented as a UML profile. However,* GASPARD2 *proposes a MARTE metamodel independent of UML in order to avoid the higher complexity handling on models. This process is presented in Chapter 5*

2. **Target-Specific Extensions**: they are all metamodels related to specific-target domain concerns, e. g. VHDL datatypes adaptation, OpenCL kernel topology.

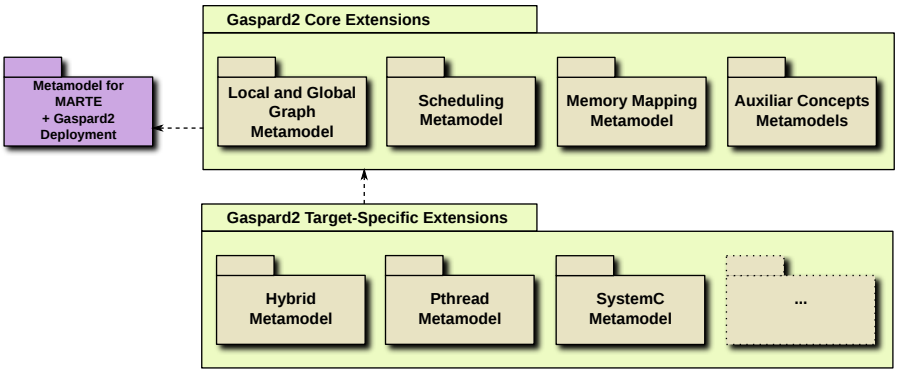


Figure 2.2: GASPARD2 Extensions for MARTE

2.2 TRANSFORMATION CHAIN

The compiling process in GASPARD2 lies in transformation chains. When we have to create a new branch, we can put various model transformations together using MDFactory (*cf.* Subsection 2.7.3) available in the framework. MDFactory takes into account all necessary procedures in order to ensure coherence among models and metamodels. This whole process is presented again in Chapter 5, however we associate the global concepts to our specific branch. Figure 2.3 summarizes the module transformation chain process. At first, we define an input model instantiating a special module "read_from_file" that reads a model stored in a file with `.uml` extension. Then, we choose the available module transformations from the *library of functionalities* according to our purposes. A special connector allows to adapt every output model from a module transformation as an input model to another module transformation.

2.3 TARGET PLATFORMS

The available versions of GASPARD2 provide support to generate code for the following platforms and languages:

- Sequential C;
- Pthread;
- OpenMP;
- SystemC;
- LUSTRE and SIGNAL;
- VHDL;
- OpenCL;

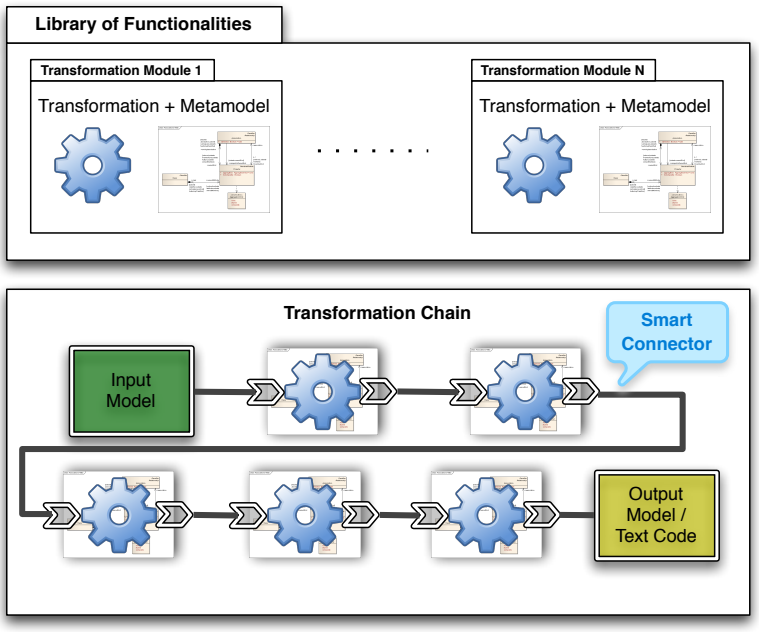


Figure 2.3: Gaspard2 Library of Functionalities and Chaining Process

We call GASPARD2 branch the set of transformation modules composing the transformation chain for a specific target. Except for the target subject of this whole thesis, OpenCL, the next subsections will present a summary of each target above listed.

2.3.1 *Sequential C*

This target is the simpler code generation. It aims at generating a code C that executes on mono-processor architectures. The generated code can be compiled and executed on a general-purpose processor regarding the execution order defined at model-compilation time according a simple scheduling policy based on data-dependence.

2.3.2 *Pthread*

In shared memory multiprocessor architectures, such as SMPs, threads can be used to implement parallelism. Historically, hardware vendors have implemented their own proprietary versions of threads, making portability a concern for software developers. For UNIX systems, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard [83]. Implementations that adhere to this standard are referred to as POSIX threads, or Pthreads [109]. For many architectures, including Windows, open source implementations exist.

The Pthread support in Gaspard2 allows generating code according to POSIX threads. Indeed, tasks or their repetitions can be split into many threads as possible in order to take advantage of multiprocessor architectures. In this case, each core run one or more threads. It is a responsibility of the model transformation chain to create the partitioning of the application's tasks into different threads in the system. Obviously, at a higher-level model, the model designer can express the intended partitioning. Otherwise, the model compiling process can define a default one.

2.3.3 *OpenMP (Fortran and C)*

This is a result from the Julien Taillard's thesis [137] and it is described with more details in Chapter 1. In summary, this branch allows for code generation for parallel applications in a shared memory model.

2.3.4 *SystemC*

There are Verilog, VHDL and a lot of other languages to implement and simulate hardware. The main issue to introduce a new language, is the increasing design complexity of systems. Like Verilog and VHDL, SystemC [72] also supports modeling at Register Transfer Level, however, the major reason for using it, is to work on a higher abstraction level like Transaction-Level-Modeling (TLM). TLM is an intermediate level which is abstract enough to allow complete system architecture design, while being accurate enough to allow performance analysis.

The code generated by this branch allows to simulate embedded systems in SystemC [124, 9].

2.3.5 *LUSTRE and SIGNAL*

LUSTRE [23] is a synchronous data-flow language for programming systems which interact with their environments in real-time. The main application field is the programming of automatic control and signal processing systems. For LUSTRE, a program is a system of equations defining variables, which are functions from time to their domain values. Since LUSTRE is concerned with discrete systems, time is projected onto the set of natural numbers, making variables infinite sequence of values. SIGNAL [74], just like LUSTRE, is a language for synchronous systems.

This branch [146, 1] in Gaspard2 generates synchronous equations according to the modeled system/application. These equations can be, then, analyzed by formal methods.

2.3.6 *VHDL*

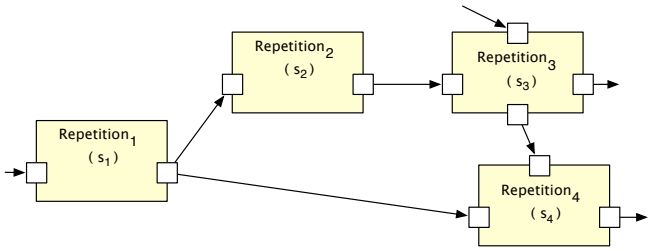
VHDL [8] is a High Description Language (HDL) for the design of integrated circuits at the Register Transfer Level. It can be used in wide range of contexts such as implementation in reconfigurable architectures, i. e. FPGAs [77]. VHDL supports modular semantics; and abstract behavioral models can hide implementation details. A part of VHDL is synthesizable and can be implemented on target FPGAs. Here, GASPARD2 generates [95, 126] VHDL code in order to create systems running on FPGAs or ASICs.

2.4 DEPLOYMENT AND IPS

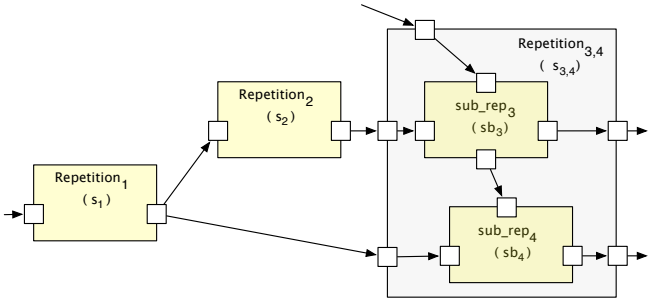
In order to generate a whole system from a high level specification, all implementation details of every elementary component have to be declared. Low level details are much better described by using usual programming languages instead of graphical UML models. In the SoC industry, individual components are called Intellectual Property (IP). IPs are also used to ease component reuse. They correspond to one specific implementation of a given functionality, either hardware or software. In SoC design, one functionality can be implemented in different ways. This is necessary for testing the system with different tools, or at different abstraction levels. For instance, different IPs can be provided for a given application component and may correspond to an optimized version for a specific processor or a version compliant with a given language. Although the notion of deployment is present in UML, the SoC design has special needs, not fulfilled by this notion. Hence, Gaspard extends the MARTE profile to allow deploying elementary components with IPs. For this purpose, it was introduced the concept of VirtualIP to express the behavior of a given elementary component (either software or hardware), independently from the usage context. A VirtualIP is implemented by one or several IPs, each one being used to define a specific implementation at a given abstraction level and in a given language. Finally, the concept of CodeFile is used to specify, for a given IP, the file corresponding to the source code and its required compilation options. The used IP is selected by the SoC designer by linking it to the elementary component through the Implements dependency. Some IPs provided by the SoC industry can be parametrized. These parameters are specified using the Characteristic concept.

2.5 MODEL REFACTORING

Gaspard2 provides resources to refactor models aiming to a better generated code. For the time being, these resources are limited to ARRAYOL transformations [62]. ARRAYOL transformations have a deter-



(a) Chain of repetitions example.



(b) After an initial fusion operation.

Figure 2.4: An example of refactoring

minant role on improving MARTE based models. They can be used not only for optimization but also as a tool for refactoring the application. Similarly to loop transformations, ARRAYOL transformations has been designed to allow the adaptation of the application to the execution, allowing to choose the granularity of the flows and a simple expression of the mapping by tagging each repetition by its execution mode: data-parallel or sequential. Currently it is just an instrument in the hands of the designer but in the future, when the necessary concepts will be introduced to ARRAYOL, optimization algorithms using these transformations will be designed and implemented. These optimizations also depend on the execution model chosen for the ARRAYOL model and they might evolve in parallel with the evolution of the execution models.

The loop transformation technique aims at improving the data access regularity and locality and removing the system-level buffers of the application codes.

For instance, Figure 2.4 gives us an overview of model refactoring in Gaspard2. Basically it implements at high model level operations aiming at improving the way how the model was designed. Here, we have a simple repetition fusion example: $\mathtt{Repetion}_3 \oplus \mathtt{Repetion}_4$. Figure 2.4a has 4 repetitions and we intend to fuse the repetitions 3 and 4 seen in Figure 2.4b. Mathematical analysis can transform the repetition spaces s_3 and s_4 into another repetition space $s_{3,4}$, i. e., $s_{3,4} = \mathrm{LCM}(s_3, s_4)$ for example. This operation, however, requires modifying s_3 and s_4 in order to ensure the same local repetition space. In this case, $sb_3 = s_{3,4}/s_3$ and similarly sb_4.

2.6 TRACEABILITY

Traceability is the ability to establish degrees of relationship between two or more products of a development process, especially products having a predecessor/successor or master/subordinate relationship to one another [61].

The connection of source and target models in a model transformation is called tracing. It establishes links between a source model element and its corresponding target element. Not all model transformation languages offer built-in support for tracing. In most languages, there is not support to it. However, tracing can be implemented by the developer. Even if a model to model transformation tracing exists, it may lack a whole global trace for a given transformation chain.

To solve some problems that deal with compilation analysis (e. g., system debugging, transformation debugging, design alternative exploration), Gaspard2 has defined its own trace approach [63]. This approach provides a traceability locally and globally. It relies on two metamodels: the Local Trace metamodel corresponding to the model to traceability and the Global Trace metamodel helping in the global navigation. The proposed mechanism is based on traceability to locate errors in a single model transformation or a transformation chain. This reduces the investigation field to the rules called to create an output element identified as erroneous in a preliminary test phase. The localization is based on three main parts, errors observed in an output model, our trace models and the localization algorithm. The errors can be pointed out by an oracle whereas the traces give the support for the localization algorithm. As the algorithm is based on generic trace metamodels, it is language independent and can be reused for any transformation languages as long as the local and the global trace are generated. However, current approaches were written in QVTO and that kept compatibility with the whole implementation. Furthermore, it has also been successfully tested on transformations using a dedicated Java API. Figure 2.5 summarizes the application of traceability on Gaspard2 models. Even if having trace information locally stored in each single model transformation (*Local Trace* in the figure), a global trace (represented by a set of chained *Local Model* and *Trace Model*) retains the necessary information which allows us to navigate by the entire transformation chain.

We have taken advantage of the traceability defined in Gaspard2 when we need to optimize applications by using profiling tools. Further details about this optimization is seen in Section 6.3.

2.7 RELATED TOOLS

As framework inspired on MDE, Gaspard2 has its basis founded on other tools.

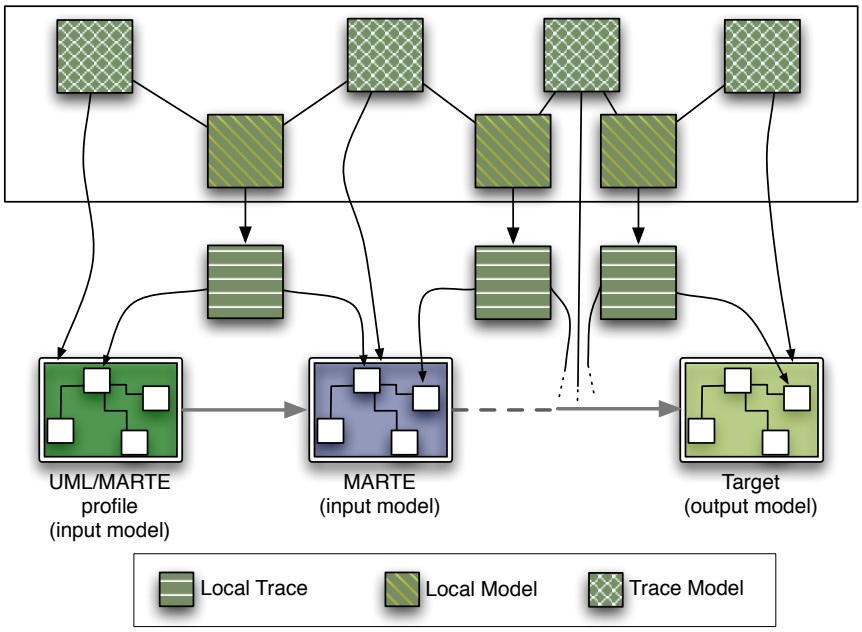


Figure 2.5: Global Tracce: Traceability Approach on Gaspard2

2.7.1 *Eclipse*

Eclipse [51] is a multi-language software development environment comprising an integrated development environment (IDE) and an extensible plug-in system. It is written mostly in Java and can be used to develop applications in Java and, by means of various plug-ins. Eclipse provides a modeling platform. Indeed, the modeling project contains all the official projects of the Eclipse Foundation focusing on model-based development technologies. They are all compatible with the Eclipse Modeling Framework created by IBM. Those projects are separated in several categories: Model Transformation, Model Development Tools, Concrete Syntax Development, Abstract Syntax Development, Technology and Research and Amalgam.

2.7.2 *Papyrus Modeling Tool*

Replacing other commercial UML graphical modeling tools, e. g., Magicdraw UML[7] , Papyrus[8] allows us to create high level UML models taking advantage of included profiles, such as MARTE. It is maintained by CEA as an opensource tool. For practical usage, Gaspard2 suggests two diagrams for model designing. On one hand, the **Composite structure diagram** shows the internal structure of a class and the collaborations that this structure makes possible. This diagram is used for designing the application itself, its target architecture and

[7] www.magicdraw.com

[8] www.papyrusuml.org

allocations. On the other hand, the **Deployment diagram** allows to model the physical deployment of artifacts on nodes.

An artifact in UML is the specification of a physical piece of information that is used or produced by a software development process and deployed on computational resources or nodes.

2.7.3 *MDFactory*

MDFactory is a Model Driven Engineering environment to design, develop and run software production chains. This tool supports all GASPARD2 branches and it is based on localized transformation. It provides a graphical editor to build such production chains with drag and drop from a reusable transformation modules library. MDFactory is based on the Eclipse platform and the Eclipse Modeling Framework (EMF). In summary, it is the tool used to build all GASPARD2 integrated transformation chains and it was originally developed by the DaRT Team[9] .

[9] *Currently, MDFactory is part of a technology transfer project that aims at spreading its use in other areas of development.*

2.7.4 *QVTO*

QVT-Operational is an imperative language designed for writing unidirectional transformations. The Operational QVT project aims to be fully compliant with the Query/View/Transformation (QVT) standard [73] providing a powerful Eclipse IDE with feature-rich editor (code completion, outline, navigation, etc.), project builders, launch configurations, deployment facilities and Ant support.

QVT and QVTO are explained with more details in Appendix B. QVTO adopts the Operational Mapping Language (OML) as extension to Object Constraint Language (OCL) with side effects that allow a more procedural style, and a concrete syntax that looks familiar to imperative programmers.

2.7.5 *Acceleo Code Generation*

Acceleo [113] is an open source code generator of the Eclipse Foundation that allow people to use a model driven approach to build application from models. It is an implementation of the standard from the Object Management Group (OMG) for model to text transformation named MOFM2T (*cf.* Section B.4).

2.8 CONCLUSION

In this chapter, we presented the reasons behind our choice of GASPARD2 as framework to implement our approach. We emphasized its other branches focused in code generation and the use of MARTE as main profile to specify software and hardware of a whole application. Moreover, GASPARD2 provides essential tools to create a transformation chain that is our backbone for the model compiler. The remaining sections deal with the basic concepts associated to this framework. In the next chapter, we present two application examples under the point of view of the model designer in a functional code generation approach towards OpenCL.

Part II

METHODOLOGY APPROACH

3

DEVELOPING APPLICATIONS

Chapter Contents

In this chapter, we attempt to clarify the proposed methodology to develop OpenCL applications from high level models. Indeed, after presenting related works and Gaspard2 as a framework for the approach, we here analyze two examples that use MARTE to specify them. Moreover, for situations where MARTE does not propose any resource, we describe the strategies to add the missing functionalities, essential to the approach.

Real world examples are appropriate to express the potential of the code generation. Additionally, they facilitate the understanding of some issues about key concepts that we propose, seen in next chapters of the Second Part of this thesis. Before introducing the main case study (described in Third Part), we provide two generic application examples from numerical method and image processing domains. For each example, we try to put in evidence the point of view of the model designer actor (*cf.* Subsection A.4.2.5) to whom the generation process is transparent. Further, we show also the point of view of the methodology provider actor when we explain the generation process.

3.1 INTRODUCTION TO MODELING METHODOLOGY

We can divide the methodology into two views. First, the global view where we define the model according to Figure 3.1. Second, the detailed view of each sub-process.

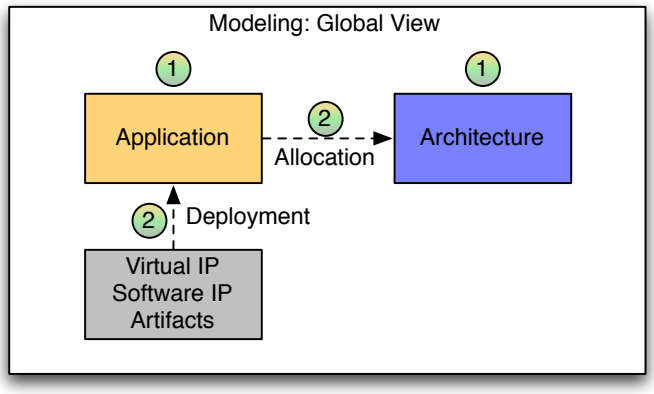


Figure 3.1: Model Creation Process : Global View

Globally, the model designer follows the steps seen in Figure 3.1 as described below. This phase is based on directives defined by MARTE and GASPARD2.

1. Designer defines application and architecture models. At first, there is no link between both models. Thus, it is possible to divide this step into two parts executed by different teams or experts.
2. Designer places every task and data onto hardware architecture elements. Moreover, at this moment the designer associates Intellectual Property (IP) to each elementary task in the application model.

We describe the detailed view in the following sections. These sections show two full examples.

3.2 MATRIX MULTIPLICATION

The matrix multiplication is an application found in most part of scientific numerical methods. For this reason and due to the high parallelism attained in this kind of application, we decided to create it as the first example for the overall approach. We have designed two different models for this application. The first one (*cf.* Figure 3.2 and Listing 3.1[10]) is the usual way in which the algorithm operates on individual rows and columns of each matrix. For the second one (*cf.* Figure 3.3 and Listing 3.2), however, we took into account performance improvements. In order to achieve the necessary reuse of data in local memory, researchers have developed many new methods for computation involving matrices and other data arrays [48, 57, 135]. Typically an algorithm that computes individual elements is replaced by one that operates on blocks of subarrays of data. The operations on blocks retain the usual way. The advantage of this approach is that

[10] *Although it is possible to use* $n \times m$ *generic matrices, in order to simplify operations, we have chosen to use* $n \times n$ *square matrices.*

the small blocks can be moved into the fast local memory and their elements can then be repeatedly used.

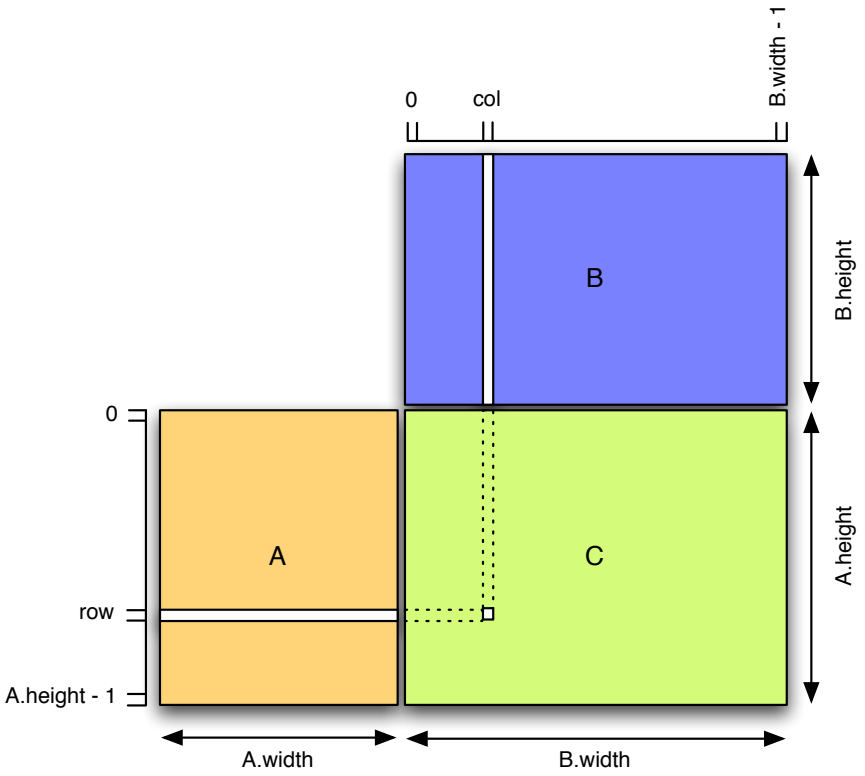


Figure 3.2: Matrix Multipliation without shared memory. *Adapted from OpenCL Programming Guide* [111]

Listing 3.1: Usual Matrix Multiplication Program

```
for (i=0; i<n; i++)
 for (j=0; j<n; j++)
  for (k=0; k<n; j++)
    C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

The entire computation involves $2n^3$ arithmetic operations, but produces and consumes only data values. Observing the algorithm, we can see that the computation presents a large reuse of data. In general, however, an entire matrix will not fit in a small local memory. The work must, therefore, be broken into small chunks of computation, each of which uses a small piece of the data. Moreover, for each iteration of the outer loop (i. e., for a given value of i) n^2 operations are done and n^2 data is re-read. For fixed values of i and j, n computation and n data is re-read too again. Hence, this does not take advantage of data reuse.

Now consider a blocked matrix-multiply algorithm presented in Listing 3.2.

Listing 3.2: Block Matrix Multiplication Program

```
for (i0=0; i<n; i0 + BLOCKSIZE)
 for (j0=0; j<n; j0 + BLOCKSIZE)
  for (k0=0; k<n; j0 + BLOCKSIZE)
    for (i=i0; i< min(i0 + BLOCKSIZE-1,n); i++)
     for (j=j0; j< min(j0 + BLOCKSIZE-1,n); j++)
       for (k=k0; k< min(k0 + BLOCKSIZE-1,n); j++)
         C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

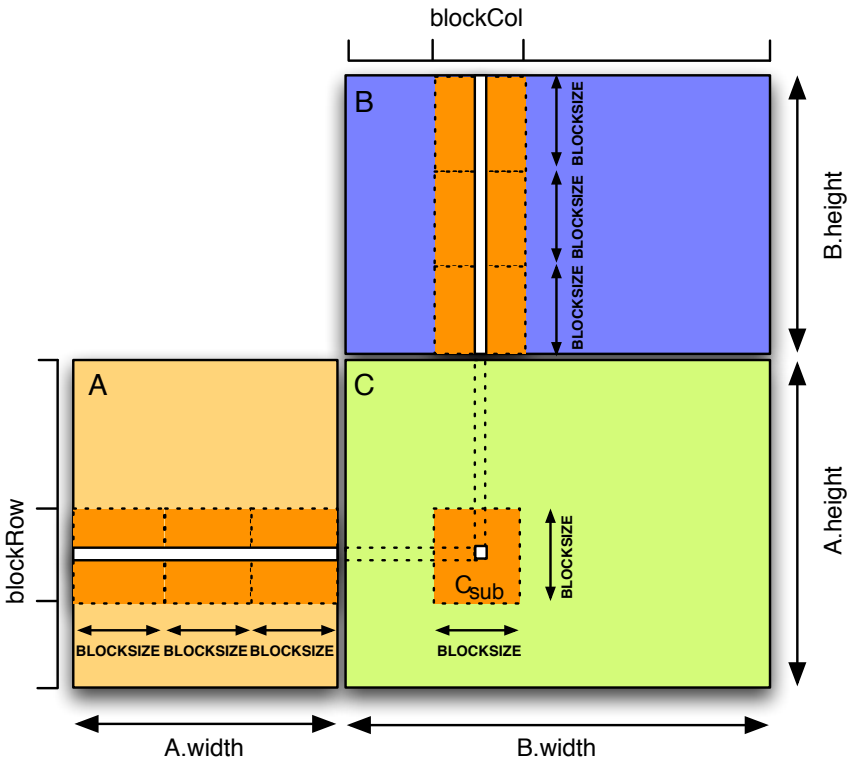


Figure 3.3: Matrix Multiplication by blocks using the shared memory. *Adapted from OpenCL Programming Guide* [111]

First, exactly the same operations are done on the same data in this program; even round-off error is identical. Only the sequence in which independent operations are performed is different from the unblocked program. There is still reuse in the whole program of order n. But if we consider one iteration with fixed i0, j0, and k0, we see that 2BLOCKSIZE^3 operations are performed (by the three inner loops) and 3BLOCKSIZE^2 data are referred to it. Now we can choose BLOCKSIZE small enough so that these 3BLOCKSIZE^2 data will fit in the local memory and thus achieve *BLOCKSIZE*-fold reuse.

In the first one, every work-item (in an OpenCL program) takes one row and one column from A and B matrices and produces one point in C matrix using scalar product. The second one is more complex; we take advantage of the organization of work-items in work-groups that perform the multiplication by block. The idea behind this approach is to take advantage using shared memories within work-groups.[11]

[11] *Remind: besides, work-items synchronization is only possible within work-groups.*

3.2.1 *Modeling the Matrix Multiplication*

To model this application, we create the diagrams enumerated below. Although all elements designed in *composite diagrams* can share a unique diagram, this division provides a better organization. The order below does not necessarily represent the sequence of activities. However, after some reflections and elaborated models, we conclude that arranging at first the basic elements, then creating compositions and their relationships, and finally linking their artifacts in different diagrams lead to a clean and organized final model.

1. **Elementary Tasks** (composite diagram). This comprehends all classes used as elementary tasks in the application.
2. **Application** (composite diagram). This comprehends the application itself. Here we design all *compound component* and their relationships.
3. **Architecture** (composite diagram). Depending on the application complexity, here we specify our available running hardware architecture. For this case and most cases, this diagram is just a generic host CPU and device GPU separation.
4. **Task Allocation** (composite diagram) comprehends the association (allocation) between tasks designed in the Application diagram and processor elements designed in the Architecture diagram.
5. **Memory Allocation** (composite diagram). Similarly to previous diagram, this comprehends the association between *FlowPorts* (data elements) from the Application diagram and memory elements designed in the Architecture diagram.
6. **Pre-Deployment** (composite diagram). This diagram contains the association between elementary tasks and their *virtual IPs* and *software IPs* as introduced in Chapter 2.
7. **Library** (deployment diagram). Here, physical IPs (external files) are associated to special classes representing *software IPs*.

3.2.1.1 *Defining the Elementary Tasks*

In Figure 3.4, we present a composite diagram for the elementary tasks. We define 4 elementary tasks as described below:

- **MA_Gen** represents a single task that produces an output data port of 4096x4096 elements of the type *real*. This task creates randomly[12] all elements of the matrix A.
- **MB_Gen** performs the same **MA_Gen**'s job[13] for the matrix B.
- **MB_Print** is the class responsible for displaying or writing results (the input data port) to a file.
- **Multiply** makes the dot product of two vectors of 4096 elements each.

[12] *We have chosen random elements, but it can also be elements read from a file, for instance.*

[13] *Although MA_Gen and MB_Gen run the same operation, for the time being we have to declare both classes due to technical constraints in UML.*

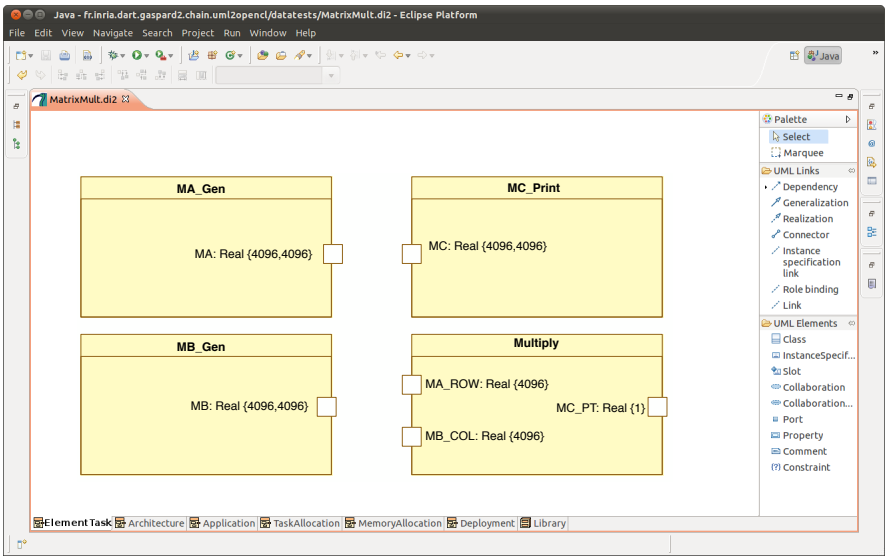


Figure 3.4: Elementary Tasks in the Eclipse Environment with Papyrus Modeling Tool

Next step comprehends the use of those elementary tasks to compose other major tasks in a higher hierarchy. Further, we establish the relationship among their ports.

3.2.1.2 *Defining the Application*

Analyzing Figure 3.5, initially we create the composed task *MultiTask*. It consists of $\begin{pmatrix} 4096 \\ 4096 \end{pmatrix}$ or, using the MARTE notation, $\{4096, 4096\}$ (defined with the shape stereotype) repetitions of the elementary task *MA_Gen*, one repetition for each point in the resulting matrix. Globally, the *MultiTask* receives two complete matrices in its input ports (A and B) and produces another complete matrix in its output port (C). As we connect ports with different shapes, we apply the *tilers* stereotypes on each connector between the composed task ports and the repeated task's ones.

The shape is a special stereotype from the MARTE profile that allows us to specify the multiplicity of an element under the point of view of the Repetitive Structure Modeling (RSM) package

Tilers are part of ArrayOL and we provide further explanation in Subsection B.2.2 of Appendix B. The specifications of *tilers* for this example are as following:

INPUT TILER A origin=$\begin{pmatrix}0\\0\end{pmatrix}$, paving=$\begin{pmatrix}0 & 0\\0 & 1\end{pmatrix}$, fitting=$\begin{pmatrix}1\\0\end{pmatrix}$,

where $\begin{pmatrix}0\\0\end{pmatrix} \leqslant \mathbf{r} < \begin{pmatrix}4096\\4096\end{pmatrix}$ and $\begin{pmatrix}0\end{pmatrix} \leqslant \mathbf{i} < \begin{pmatrix}4096\end{pmatrix}$.

INPUT TILER B origin=$\begin{pmatrix}0\\0\end{pmatrix}$, paving=$\begin{pmatrix}1 & 0\\0 & 0\end{pmatrix}$, fitting=$\begin{pmatrix}0\\1\end{pmatrix}$,

where $\begin{pmatrix}0\\0\end{pmatrix} \leqslant \mathbf{r} < \begin{pmatrix}4096\\4096\end{pmatrix}$ and $\begin{pmatrix}0\end{pmatrix} \leqslant \mathbf{i} < \begin{pmatrix}4096\end{pmatrix}$.

OUTPUT TILER origin=$\begin{pmatrix}0\\0\end{pmatrix}$, paving=$\begin{pmatrix}1 & 0\\0 & 1\end{pmatrix}$, fitting=$\begin{pmatrix}0\end{pmatrix}$,

where $\mathbf{r}$ is the same as input tilers and $\begin{pmatrix}0\end{pmatrix} \leqslant \mathbf{i} < \begin{pmatrix}1\end{pmatrix}$.

The remaining application consists in interconnecting all earlier defined tasks. In summary, the composed task *Application* has an instance *mtask* of the *MultTask* that receives a matrix nxn from the instance *magen* by a simple connector. We do the same operation for *mbgen*, and finally, the task instance *mcprint* prints out the resulting matrix. Specially for illustrating this example, we define a loop of 5 iterations for the whole application and insert a mandatory general application instance (*appli*) in the model.

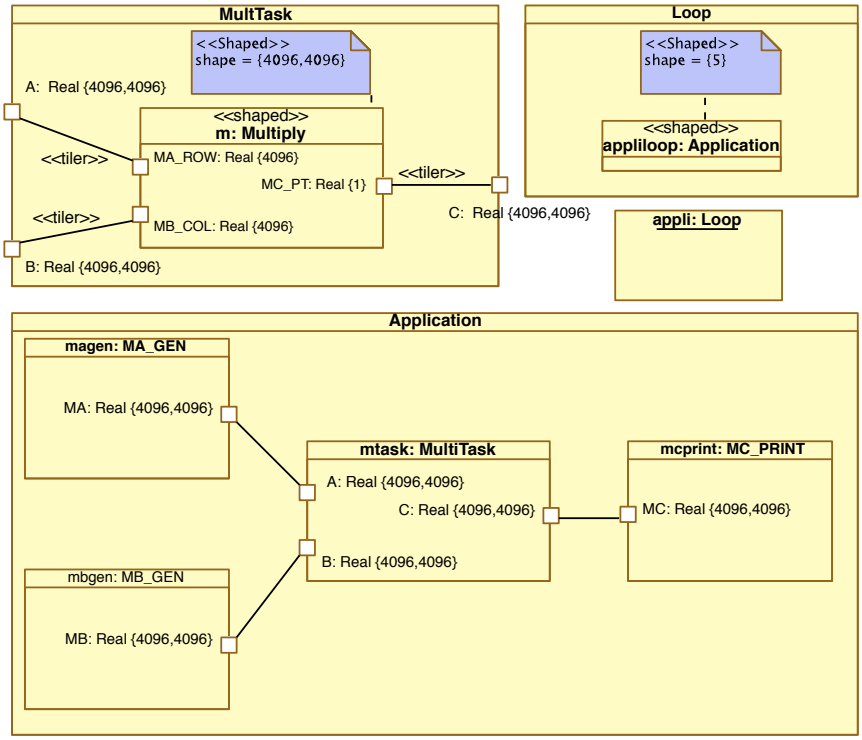


Figure 3.5: Application Model for Matrix Multiplication

3.2.1.3 *Defining the Architecture*

The architecture model for this example is simple. We do not need to express all the complexity of CPUs and GPUs in this model because the application model uses only the basic structures. This is only to keep the model simple. Nevertheless, depending on applications we can add further elements, such as *local memory* of Streaming MultiProcessors. Figure 3.6 shows the architecture model, it is composed of two *hwResource*: *host* and *device*. Both *host* and *device* have a *hwProcessor* and a *hwRAM* representing their corresponding processor[14] and memory. In order to distinguish host from device, we add a special tag description as required by the hybrid transformation module (*cf.* Subsection 5.5).

[14] *Even if the GPU is a many-core processor, here we design it as a single processor representing the whole device.*

MARTE does not provide any direct way to identify the important definition of "host" and "device" from the OpenCL programming model. To meet this need, we write this information directly on the tagged value ***description****. In Chapter 5 we present the way to process this information in order to create specific code for "host" and "device".*

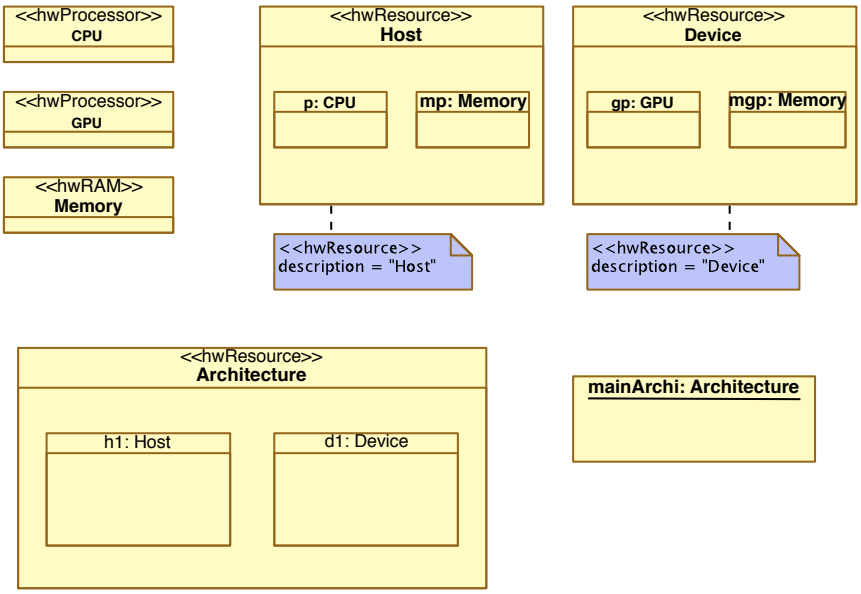


Figure 3.6: Architecture Model

3.2.1.4 *Allocating Data and Tasks*

This is one of the most decisive steps in the modeling process. Indeed, from these allocations we can identify kernels and all variables of the system, and how they relate to each other. For this application, we have 4 tasks instances: *magen*, *mbgen*, *m*, and *mcprint*. Producing and printing data in the application are tasks of the *host*. The *device* has the task of performing all operations of matrices. Consequently, as seen in Figure 3.7, the repeated task instance *m* is allocated onto the GPU(device) processor. The other ones are allocated onto the CPU(host) processor. These allocations are implemented with the dependency relationship "«abstraction»" that is stereotyped as "«allocate»".

One important feature, emphasized in the model compiling process, is the capability of using "shaped" processors. In this case, we can distribute equally the instances of repetitions of an allocated task.

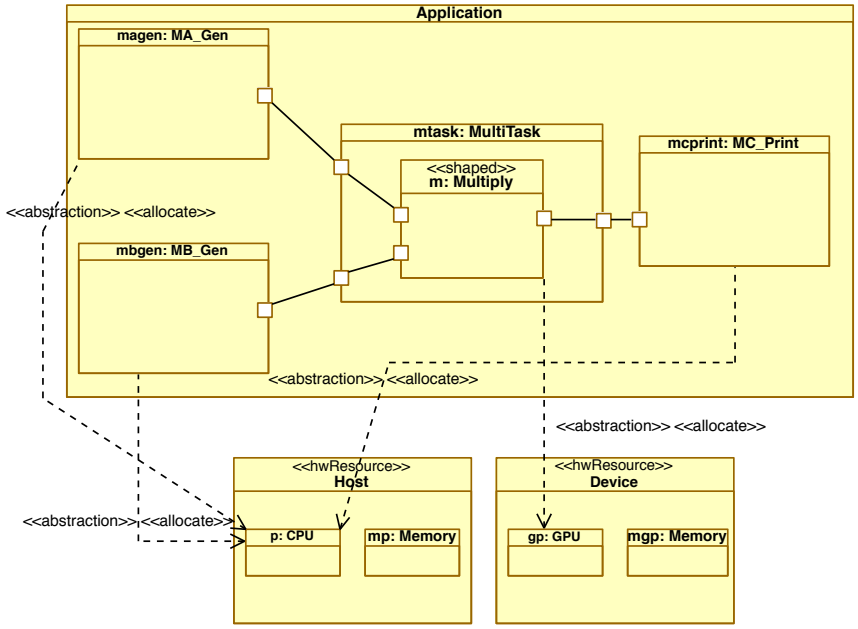


Figure 3.7: Task Allocation

Another noteworthy allocation is the data allocation. The MARTE-to-OpenCL branch of Gaspard2 is the first approach that take into account the memory distributed aspect of two independent processors. For this reason, the model designer must specify where the data will be allocated. For instance, all *flowports* of each allocated task are allocated onto the corresponding memory of each hardware resource (*cf.* Figure 3.8). Not all *flowports* have to be explicitly allocated. Indeed, some phases (*cf.* Chapters 4 and 5) in the model transformation chain are responsible for analyzing those allocations in order to optimize the memory space and data communication. As a result, two *flowports* can have the same memory address.

3.2.1.5 *Deploying Elementary Tasks*

As seen in Section 2.4, although MARTE is suitable to common modeling purposes of real-time and embedded systems, it lacks the resources to integrate from high-level modeling specifications to low-level concerns. Gaspard2 bridges this gap and introduces additional concepts and semantics to fill this requirement. The methodology uses the deployment defined in Gaspard2. In this example, *MA_Gen* and *MB_Gen* are similar tasks. Therefore, they have the same `Virtual_IP` and `Software_IP`. UML dependency connectors stereotyped with "«implements»" establish the association for all elementary tasks. Figure 3.9 presents this process.

The second part of the deployment phase is the artifacts manifestation (Figure 3.10). An artifact is a classifier that represents some

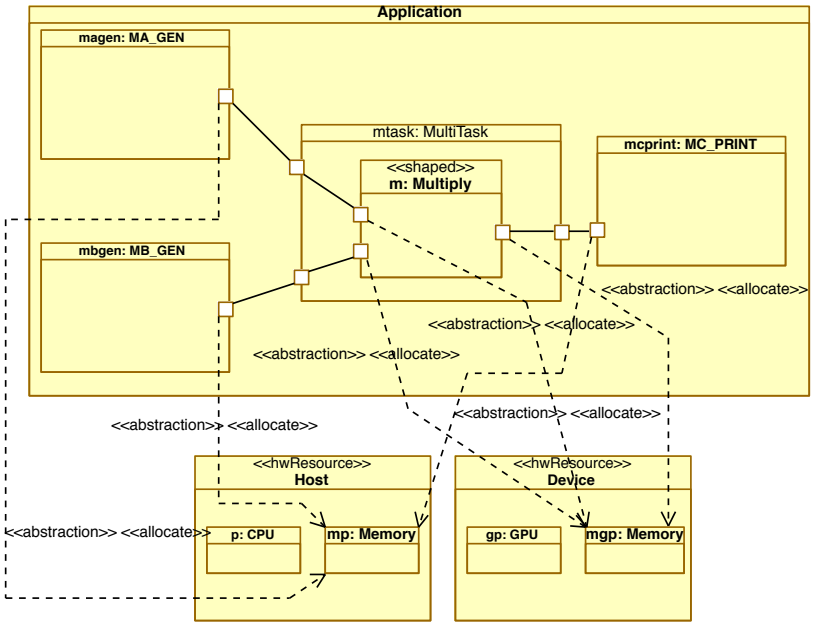


Figure 3.8: Data Allocation

physical IP (source snippet) that is used by the software development process. The artifacts are stereotyped with "«codeFile»". This stereotype has tagged values containing the source filename, header filename, compilation and link directives, and the parameter order. The IP integration is again presented in Section B.4 that deals with code generation.

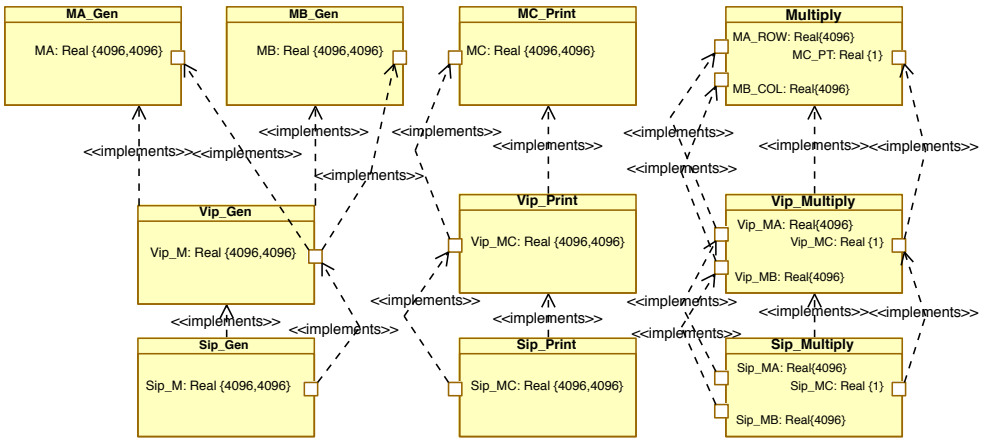


Figure 3.9: Deployment Phase: Virtual IP and Software IP

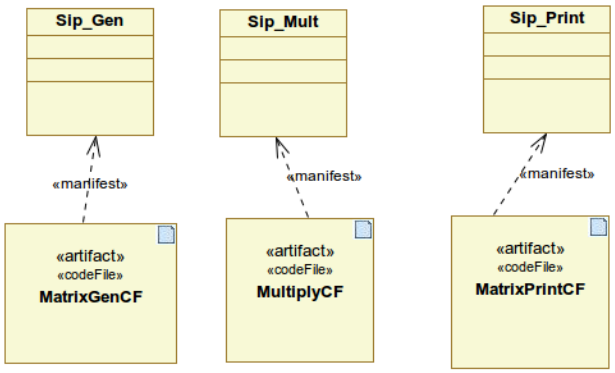


Figure 3.10: Deployment Phase: Artifacts Manifestation

3.2.2 *Generating Code*

Once all previous steps are ready, the designer starts the code generation process directly from the Eclipse interface. The internal operations in this phase are hidden from the designer.

3.2.3 *Results and Benchmarks*

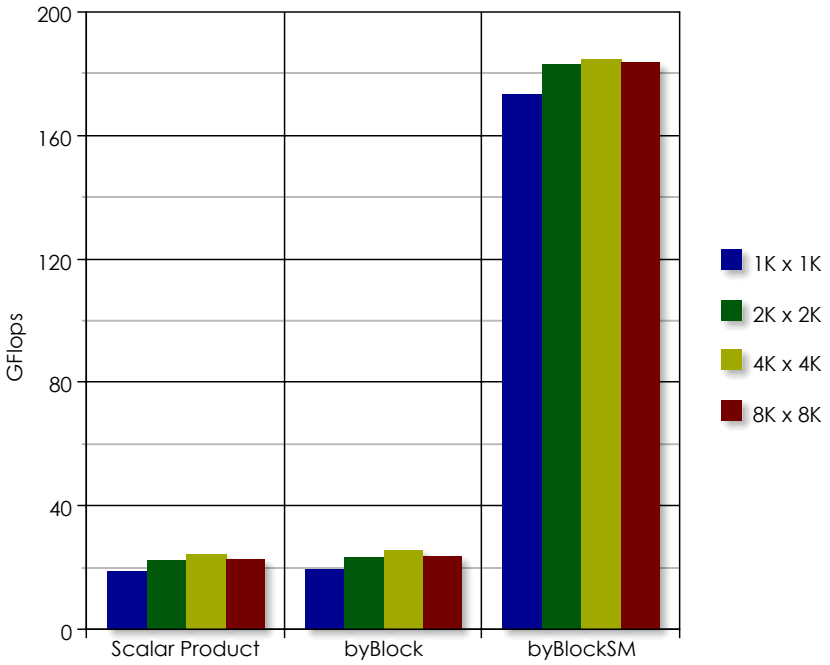


Figure 3.11: Results for Matrix Multiplication Example

Figure 3.11 illustrates three results from the three OpenCL code running on 4 different matrix sizes. The device used here is one of

the T10 GPUs of a NVidia S1070 card. There is no performance gain between the first and second ones. Indeed, even when having different models, all work-items do the same computing work. However, the third one has an expressive increasing in performance (about 8x in double precision). This one is the implementation of the blocked version of the matrix multiplication. Figure 3.12 shows the modifications (conform to Figure 3.3) of the multiplication task in the model of Figure 3.5. As seen before, the blocked version take advantage of the shared memory of work-groups (see the allocation details in the figure), and this leads performance to higher levels. The `BLOCKSIZE` is equal to 16 and each work-item goes through 256 sub-matrices in this example. Further, we allocate each sub-matrix in the shared memory.

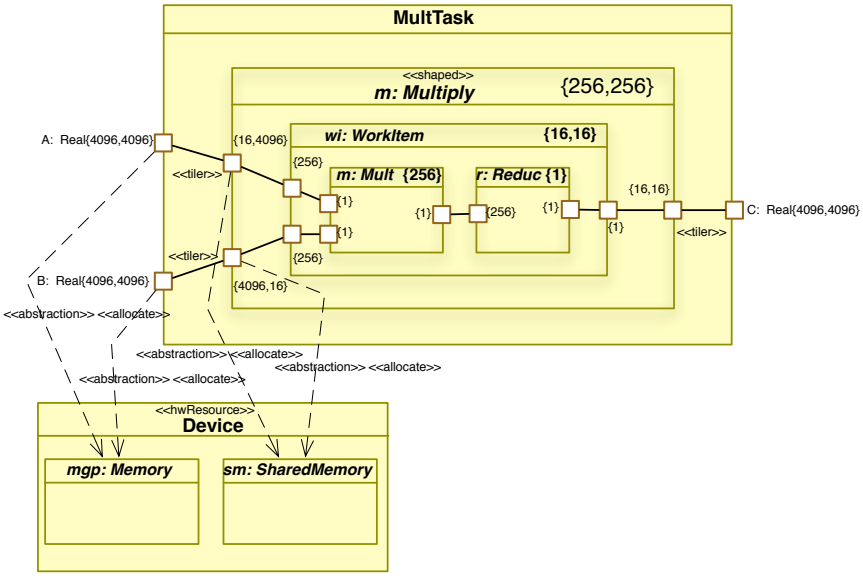


Figure 3.12: Blocked Version Model

Enabling optimization analysis in the tiler transformation, data copies from the global memory to shared(local) memory in GPU processors allow for fast data access and reuse by all the work- items of the same work-group. Additionally, based on these results, different matrix sizes do not have substantial influence on performance.

3.3 SIGNAL PROCESSING

The second case study concerned in this startup testbed of the whole approach deals with specific aspect of pre-processing to H.263-based video compression standard: scaling. This choice has a reason: intensive signal processing suits well to massively parallel architectures. Moreover GPGPU is increasingly being part of embedded systems. Thus, the scaling during video-compression is essential for previews

or for streaming for small form factor devices, such as mobile phones. The application consists of a classical downscaler which transforms a video signal, which, for instance, is expressed in 1080p or High Definition Common Intermediate Format (HD-CIF) used in HDTV, into a smaller size video. In this situation, the downscaler can be composed of two components: a horizontal filter that reduces the number of pixels from 1080-columns to 480-columns and a vertical filter that reduces the number of pixels from 1920-lines to 720-lines by interpolating packets of 8 pixels both column- and row-wise.

In a typical case of handling a 25-frames-per-second video signal lasting for 80-seconds, the downscaler may process up to 2000 frames in HD-CIF format, with each input frame being represented by a two-dimensional array of size 1920×1080 and should emit 2000 output frames of size 720×480. Since each video pixel is encoded in 24-bit RGB color model, the frame generation process is repeated for each frame and for each pixel of different color space along two different directions. The final frame is produced by using these outputs from different color space. Depending on the composing function, a broad-range of output colors are possible for each pixel and thus for each frame. The Figure 3.13 illustrates this basic operation for a given frame in high-definition format.

Figure 3.13: Horizontal and Vertical Filter Processes

As can be observed, the operations concerned with the scaling is highly parallel and repetitive. The interpolation is repeated for each frame, each pixel and for each color channel.

3.3.1 *Modeling the Downscaler*

In this subsection, we show an overview of the model for the Downscaler application. Even though all design steps seen in the Matrix Multiplication example should be made for this example, to avoid redundancies we present only the overall modeling. Different from the initial presentation in previous Subsection, the model described here transforms a video of size 352×288(CIF format) into a 132×128 one. This model is the same as the one designed to FPGA and other

platforms of Gaspard2. Except for the architecture and allocation, no change is necessary in the original model. The Downscaler itself is composed of two components: a horizontal filter that reduces the number of pixels from a 352-lines to a 132-lines by interpolating packets of 8 pixels; and a vertical filter that reduces the number of pixels from a 288-lines to a 128-lines by interpolating packets of 8 pixels as well. Figure 3.15 gives us an overview of this application and Figure 3.14 shows the elementary tasks that are arranged to create the whole application.

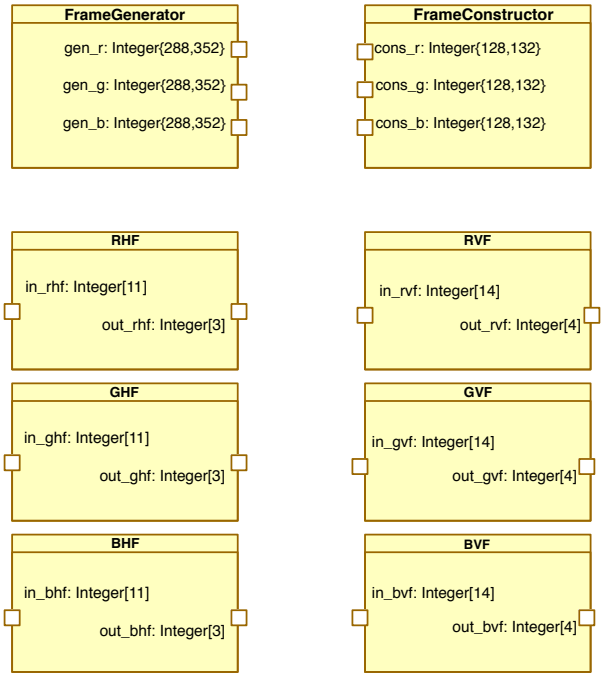


Figure 3.14: Elementary Tasks for the Downscaler

The elementary tasks comprise a task to read an input video frame, a task to save(or display) an output video frame, three tasks for each color component in horizontal filter, and three tasks for each color component in vertical filter. Although these latter tasks have same functionality, for the time being, Gaspard2 requires an elementary task for each one. Afterwards, we structure the whole application as seen in Figure 3.15. In summary, each iteration of 2000 reads a frame (*ifg*), performs the downscaling (*id*), and writes out the resulting frame(*ifc*). Moreover, the downscaling component is composed of the horizontal filter (*ihf*) and the vertical filter (*ivf*).

The composition of the horizontal and vertical filters is seen in Figure 3.16. They are composed of *shaped* tasks that perform elementary operations defined in their IPs. These tasks are potentially parallel so they are allocated onto GPUs. The defined input *tilers* specify overlapped readings and they feed each iteration with 11 pixels interleaved

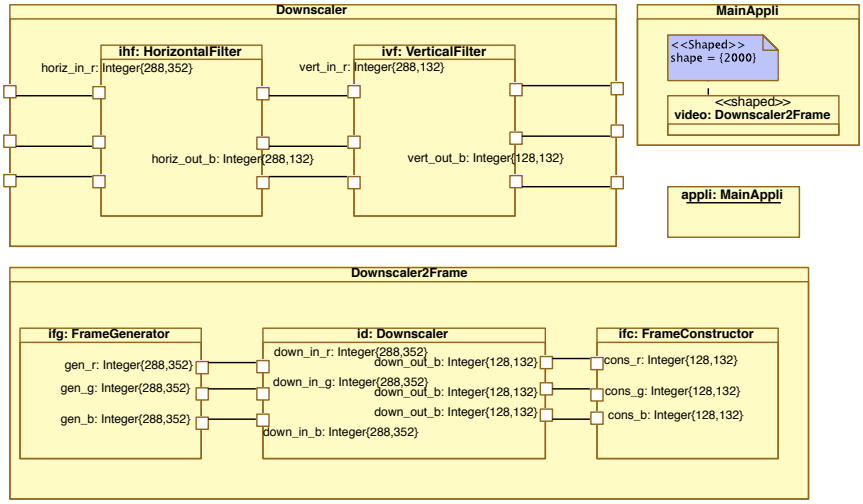


Figure 3.15: Overall Downscaler Application

by 8 pixels (horizontal) and 14 pixels interleaved by 9 pixels (vertical). The shape of tasks is appropriate to traverse the entire frame either by columns (horizontal) or by rows (vertical).

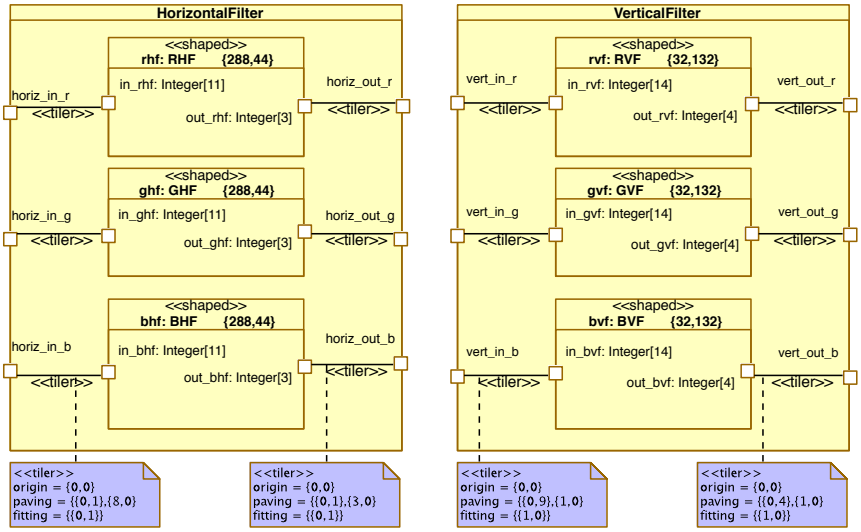


Figure 3.16: Detail of Horizontal and Vertical Filters

3.3.2 *Results and Benchmarks*

Three versions of the Downscaler were generated. Table 3.1 presents corresponding results. As expected, OpenCL versions have high speedup in relation to the sequential version running on CPU. Moreover, with the suppression of surplus memory transfers (described in Chapter 6),

we reduce the overall execution time. Indeed, in Figure 3.17, we offer profiling results that clearly show the decreasing of relative GPU times between transfers and kernel executions.

Table 3.1: Downscaler Results

Implementation	Total Time(sec)
Sequential C: this version is automatically generated by Gaspard2 with no optimization. Moreover, the CPU used is based on common desktop computers. This execution time is only to be used as a general reference and it does not represent the best performance that we can achieve in CPU based systems.	36
OpenCL not-optimized code running in one GPU T10 of the NVidia S1070 system.	4.9
OpenCL as the previous one, however we activate the memory optimization explained in Section 6.1.	3.6

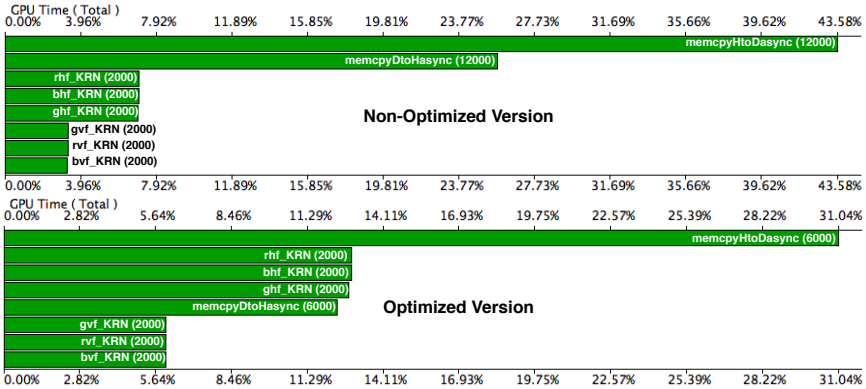


Figure 3.17: Profiling results for Downscaler Application

3.3.3 *Comparing to SAC*

In order to know how to place the efficiency of our generated code among existing code generation approaches, we provide a comparison that resulted in the paper [75]. A work made with the Compiler Technology and Computer Architecture Group of the University of Hertfordshire School of Computer Science[15] , allowed to compare our results with those produced by the Downscaler implementation for SAC. To compare the performance of the generic and non-generic Downscaler implementations in SAC, we measure the runtimes of both horizontal and vertical filters, each executed for 300 iterations on an HD video. We also compare the performance of both the sequential and CUDA code generated by the SAC compiler (denoted as SAC-Seq and SAC-CUDA respectively). The GPU used in these tests is the Nvidia Fermi GTX480. The execution times of different implementations are shown in Figure 3.18. The first point that we

[15] http://ctca.feis.herts.ac.uk

observe is that the CUDA code performs significantly better than its sequential counterpart. This is because each output frame element can be computed independently, therefore providing abundant fine-grain parallelism for GPU's massive parallel architecture to exploit. Another interesting finding is that, while execution times of sequential code do not vary significantly between generic and non-generic implementations, the non-generic filters execute 4.5x (horizontal) and 3x (vertical) faster than the generic versions on GPU. In fact, the generic output *tiler* is specified as a for-loop nest. Since the SAC compiler does not attempt to parallelize loops apart from WITH-loops, the for-loop nest is executed on the host. Since both the input tiler and task function are executed on the GPU and produce intermediate results in the GPU memory, the intermediate result has to be transferred back to the host memory before the output tiler can access it. This device-to-host transfer time significantly increases the total runtime of the filters. On contrary, the input tiler, task function and output tiler are fused into one single WITH-loop by the WLF optimization in the non-generic implementation. Therefore, it is executed on the GPU completely without any intermediate data transfers, improving performance dramatically.

3.3.3.1 *Performance Comparison of SAC and Gaspard2*

Table 3.2: Kernel execution and data transfer times of GASPARD2 implementation

Operation	**#calls**	**GPU time(μsec)**	**GPU time (%)**
H. Filter (3 kernels)	300	844185	29.51
V. Filter (3 kernels)	300	424223	14.83
memcpyHtoDasync	900	1391670	48.74
memcpyDtoHasync	900	197057	6.89
Total	-	**2.86sec**	**100.00**

Table 3.3: Kernel execution and data transfer times of SAC implementation

Operation	**#calls**	**GPU time(μsec)**	**GPU time (%)**
H. Filter (5 kernels)	300	1015137	29.60
V. Filter (7 kernels)	300	762270	22.22
memcpyHtoDasync	900	1454400	42.40
memcpyDtoHasync	900	198000	5.77
Total	-	**3.43sec**	**100.00**

Table 3.3 shows a detailed breakdown of kernel execution time and data transfer time, the non-generic SAC implementation takes to process 300 frames. Similar to our implementation (the Gaspard2 version results presented in Table 3.2), data transfers represent approximately 50% of the total execution time. The reason is that both approaches transfer the same amount of frame data to the device memory before compression starts and back to the host memory afterwards for displaying. Figure 12 shows runtime comparison between these two approaches. As we can see, horizontal and vertical filters in Gaspard2 perform slightly better than SAC. Upon further investigation, we discover that each filter in Gaspard2 is specified as a single OpenCL kernel. The final fused WITH-loop for horizontal filter after compiling has 5 generators (the vertical filter has 7 generators). Since the CUDA backend creates one kernel for each generator, this means 5 kernels have to be launched during runtime. Such large number of kernel invocations is inefficient in two aspects and causes slowdowns of the SAC implementation:

- Each kernel launch incurs context overheads. The more kernels a program executes, the higher this cost will be.
- Data in certain memory of the GPU is not persistent across different kernels, such as the on-chip L1 cache. Therefore, separating computations of the same data array into different kernels hinders effective data reuse.

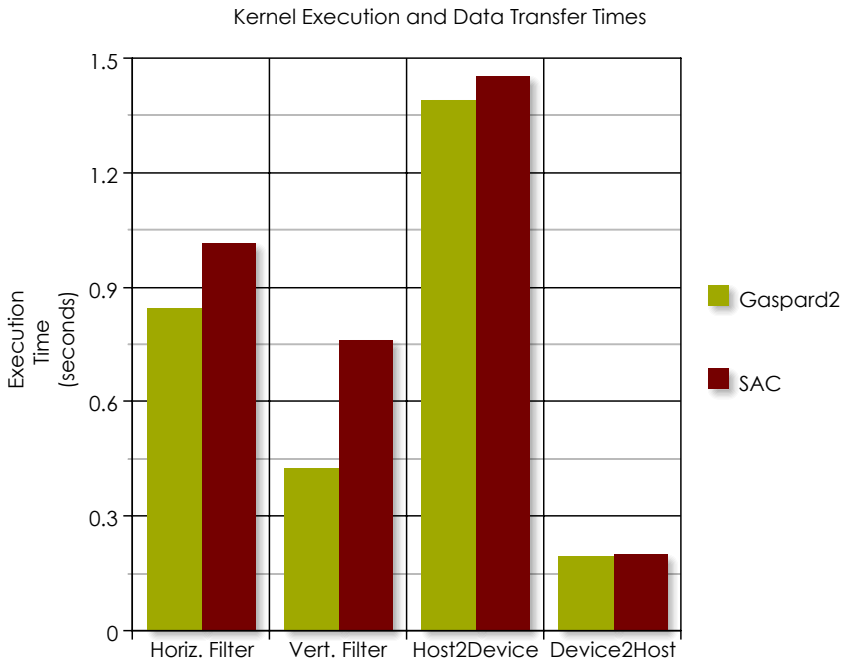


Figure 3.18: SAC versus Gaspard2 Comparison

3.4 CONCLUSION

To conclude, we remind that this chapter aims to present how the methodology works on generic applications. We illustrate all phases involved on the specification of an application and its platform. In order to generate code, designers have to specify previously their application, hardware, allocations and task deployments by associating artifacts. Through results and benchmarks, it is possible to verify the functionality and efficiency of the generated code. Further, as a validation of the efficiency relative to existing approaches, we compare our results for the Downscaler application facing the SAC implementation of the same application. This comparison shows a slight better performance for the version created by our methodology. The topics discussed in this chapter provide a basis to the following chapters that deal with model compilation.

4

METAMODELS AND GPUS

Chapter Contents

In this chapter, we begin to explain the technical process behind our methodology. At first, we clarify the main points related to principles of operation of GPUs, then we present the main metamodels that statically provide a structure that supports these principles. Therefore, to create models which express the main features found in GPU architecture and aimed application, we propose three novel metamodels. These metamodels focus on the different GPU related aspects partially presented in Chapter 3. There are other metamodels belonging to the whole approach, but they are generic technical solutions rather than a high-level description of an application designed to suit GPU architectures. They will be discussed in Chapter 5.

The following sections present the challenges found in GPU programming model, then we introduce three new proposed metamodels. The first one deals with *scheduling* of tasks aiming at defining an order of execution onto available resources. The second one aims to setup variables and their allocations into available memory banks. The third one brings up concepts closer to the target. This metamodel provides necessary elements to create key structures used in GPU programming.

4.1 METAMODELS FOR THE GPU PROGRAMMING MODEL

4.1.1 *Coprocessor*

GPUs are coprocessors (*cf.* Appendix A). Indeed, they do not have work autonomy and need a host, usually CPUs. As a graphics processor, the GPU worked already as coprocessor relieving the processor from graphics tasks. Now, as GPGPU, instead of CPUs doing the heavy parallel job, CPUs dispatch those tasks to GPUs. Although MARTE provides the resources to specify processors and their properties, it lacks specific concerns to hardware accelerators, such as GPUs. In the previous chapter, we present our methodology defining a GPU as a *Hardware Processor* with its own memory. We propose to distinctly separate CPU and GPU by defining their roles on the "description" tagged value. This is mentioned on the *Hybrid metamodel* (Section 4.4) and discussed in Chapters 5 regarding the metamodel semantics on transformations.

4.1.2 *Host and Device Memories*

Besides the complexity of the memory hierarchy on GPU, a basic structure is mandatory regarding to platform architecture: the independency between host and device memories. For the most designed applications we have to specify a host processor and a device processor containing their corresponding global memory. Sometimes we have to establish the memory hierarchy of GPU and to point out the local and private memories. This is important when the model designer decides to allocate some variables directly onto other memory address. For instance, transfers between global memory and local memory aiming at creating a designer-managed memory cache. Our methodology adopts a twofold process to handle this feature: on the one hand, the model designer specifies the platform architecture according to his intended specification level; on the other hand the *memory mapping* metamodel (Section 4.3) and corresponding transformation module create a addressing map for each memory available on the system.

4.1.3 *Work-Groups and Work-Items Topology*

In Appendix A/Figure A.7, we present a typical work-items distribution topology. Functionally, this does not have any impact on execution [1]. However, this can highly impact the final performance. In our methodology, we have decided to give the responsibility for this topology the model designer. Indeed, the tasks's shape is the information used to define the topology to launch a kernel. For convention, the first dimension of the shape represents the 1D work-group dimension and the other ones are the work-items dimensions (from

1. Except if the data access depends strongly on multidimensional indexes

1D to 3D). This process is again presented in the *Hybrid* metamodels and transformation modules.

4.1.4 *Optimizations*

We dedicate a whole chapter (Chapter 6) to discuss the points we have chosen to optimize in GPU programs. However, in the metamodels described below, we emphasize often some notions that concern to optimization aspects such as kernel launching topology and memory copies. This is mainly verified on the Hybrid and memory mapping metamodels.

4.2 SCHEDULING

One of the main issues in application design is the process of deciding how to commit resources among a variety of possible tasks. In a model based on MARTE, the task order can be observed directly from the data dependencies among tasks. The scheduling analysis may start as soon as the application and the architecture models are available. Figure 4.1 illustrates a simple application with multiple tasks. All tasks are allocated onto only one processor and they will run sequentially or concurrently according to a scheduling policy and data dependency among them. For instance, the tasks *vec1* and *vec2* in Figure 4.1 are responsible for generating the data *genarray1* and *genarray2*. Both tasks do not have any data interdependency. This means that these tasks can run concurrently according to available resources. Globally, the part **C** depends on the task *dev* (**B** composed of **B.1** and **B.2**) which depends on the tasks *vec1* and *vec2* (**A**). Regarding only the composed task *dev* that comprehends tasks *ep* and *s*, the last one cannot start before the first one finish its job. So, for this example, we can express an ordered list for execution as: {*vec1 // vec2*, *ep*, *s*, *pdot*}. This is a simple example but our approach is able to handle more complex ones and to provide a valid scheduling policy for the application. Next subsections analyze implementation key points in order to create a model for the task graph and to define an ordered list (scheduling) globally and locally for each computing resource.

4.2.1 *Building a Task Graph*

In general, a task graph is a graph in which each node represents a task to be performed. A directed arc from T_a to T_b indicates that task T_a must complete before task T_b begins. Each node generally has the form presented in Figure 4.2a. Figure 4.2b illustrates a simple example of a task graph. Here, taking T_6 as example, it begins when T_1, T_2, and T_3 complete. *Execution times* are analyzed to better manage the scheduling policy decision. Our approach, however, does not yet take

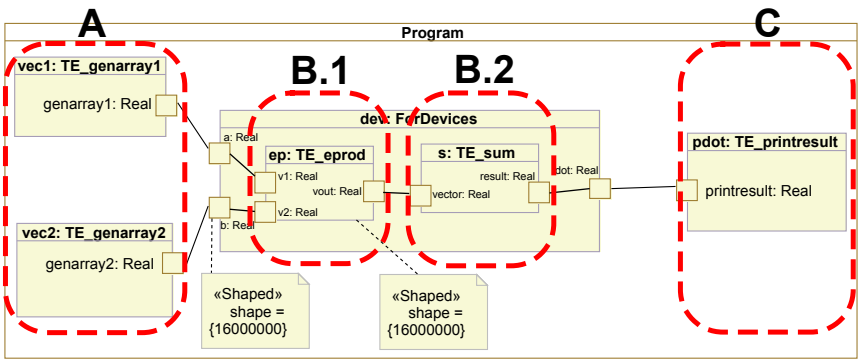
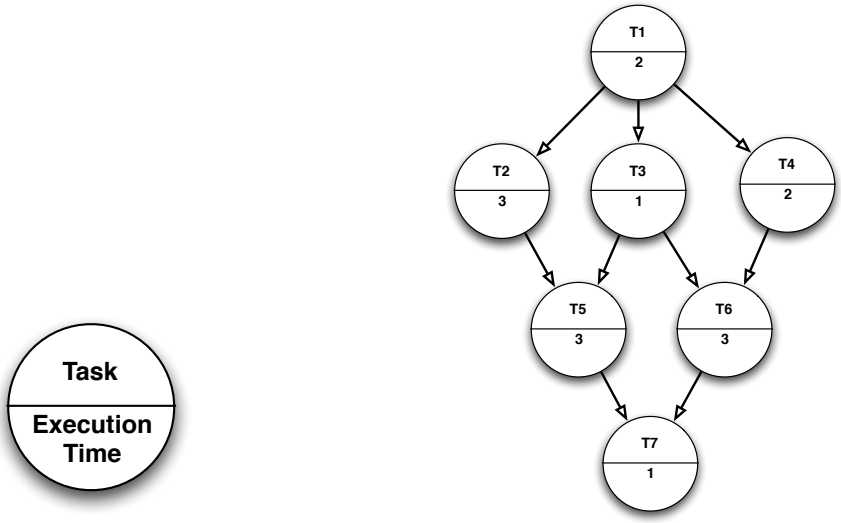


Figure 4.1: Tasks and Allocation



(a) Graph node form.

(b) Simple graph example.

Figure 4.2: General form of task graph representation.

into account the *execution time* of tasks and uses task graphs to allow some algorithm making static allocations of tasks to the processors.

4.2.1.1 *Local Task Graph*

The metamodel proposed for definition of task graph at the first level of each composed component in an application model (e. g., *Program* and *dev:ForDevices* in Figure 4.1) is depicted in Figure 4.3. In this metamodel, the focus is on creating a task graph for a structured component such as composed components. For instance, analyzing a model conforming to this metamodel we can verify:

- for a structured component **sc**, defined in an application by using UML/MARTE, is created one graph **gr** contained by and referenced to **sc**;

- the graph **gr**, in its turn, can be composed of several nodes (representing tasks) which can have dependencies among themselves;
- the dependencies can be *intra*, when the task depends on tasks allocated onto the same processor, and *extra*, when it depends on tasks allocated onto different processors;
- nodes are associated to *AssemblyParts* (i. e., instances of task classes) and *Distribute* connectors which are used in task allocations onto processors.

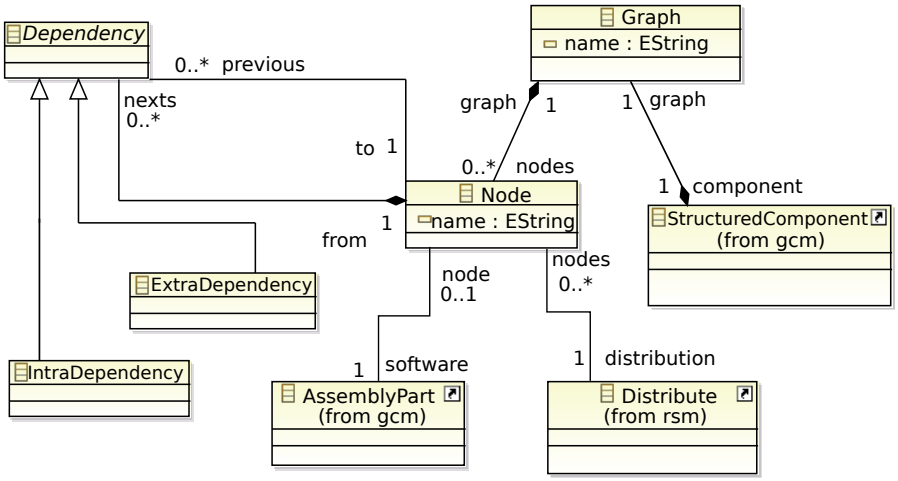


Figure 4.3: Local Task Graph Metamodel

An example of model produced by the *Local Task Graph* metamodel is showed in Figure 4.4. The figure emphasizes the node *dev* containing three arcs as *IntraDependency* elements. Each arc has two endpoints represented here by the references *to* and *from* which allow to identify their directions.

Notice that each *structured component* in the application model must have a task graph model, even components that have no hierarchy level. This is necessary to construct the global task graph from the many individual local task graphs.

4.2.1.2 *Global Task Graph*

The global task graph organizes the graphs of each structured component of the application model. The idea behind the global graph is gathering all sub-graphs previously created into one single hierarchic structure whose elements reference actions in the whole application. Figure 4.5 shows the metamodel proposed to implement the global task graph. The main graph belongs to an application *instance* and its nodes, differently from the local graph, are called *tasks*. The class *Task* is an abstract class and has 5 derived classes which are classifiers for tasks. Here they are:

- **GlobalGraph** points to task graphs that gather information at lower hierarchy levels;

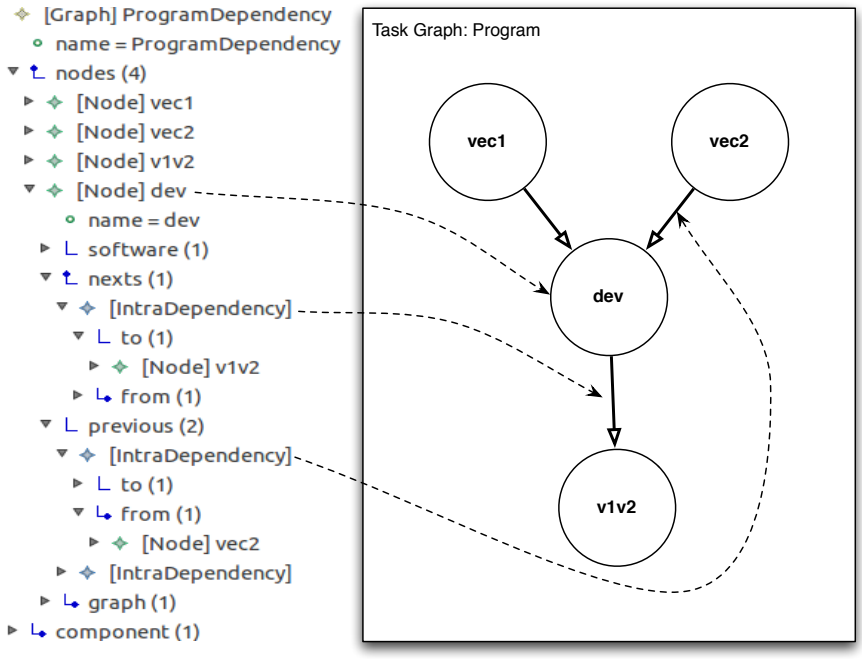


Figure 4.4: XMI Model Sample conforming for Local Task Metamodel and its Representation

- **StartTask**: is a special virtual task defining an entry point to the main graph or sub-graphs;
- **EndTask**: is like the previous task, however indicating the exit point;
- **IPTask**: is a task directly associated to elementary tasks in the application;
- **TilerTask**: is a special task usually not defined in application but derived from tiler connectors.

Similarly to local graph, tasks can be associated to one *AssemblyPart* and/or one *Distribute* connector. Moreover, dependencies among tasks themselves allow to express the relationship to define a later scheduling. These dependencies can be defined as *TaskIntraDependency* when co-related tasks share the same processor and *TaskExtraDependency* when they share independent processors.

Figure 4.6 illustrates a result model example created from the previously depicted metamodel. This example is well suited to show the levels of the graph hierarchy. The first level contains only three tasks: *StartTask*, *Global Graph p1_Task*, and *EndTask*. However, the middle task contains another graph itself, so we can observe the second level in the hierarchy. Although, the XMI file sample (left side in Figure 4.6) does not show all attributes (such as *next* and *previous*) that allow to see the dependency relationship between tasks, it contains all necessary data

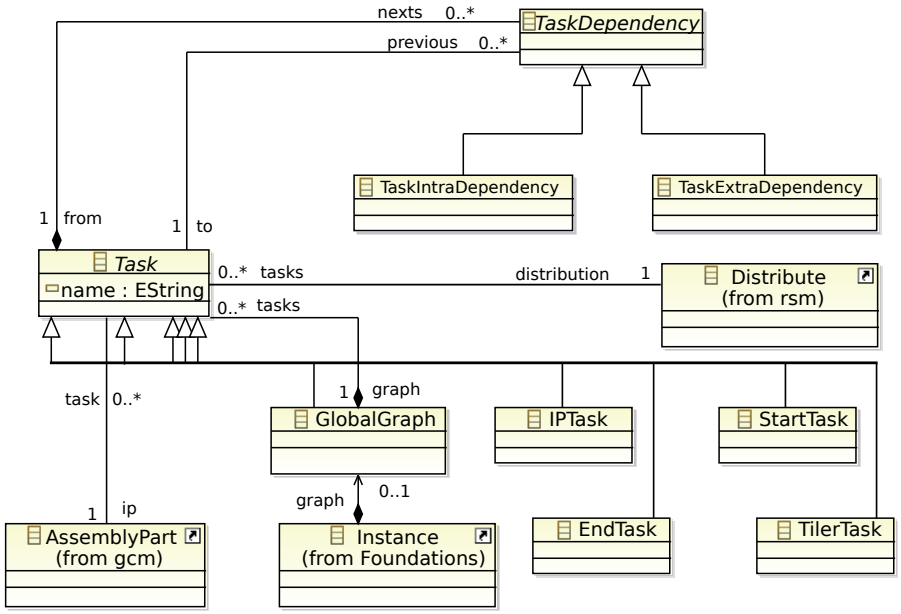


Figure 4.5: Global Task Graph Metamodel

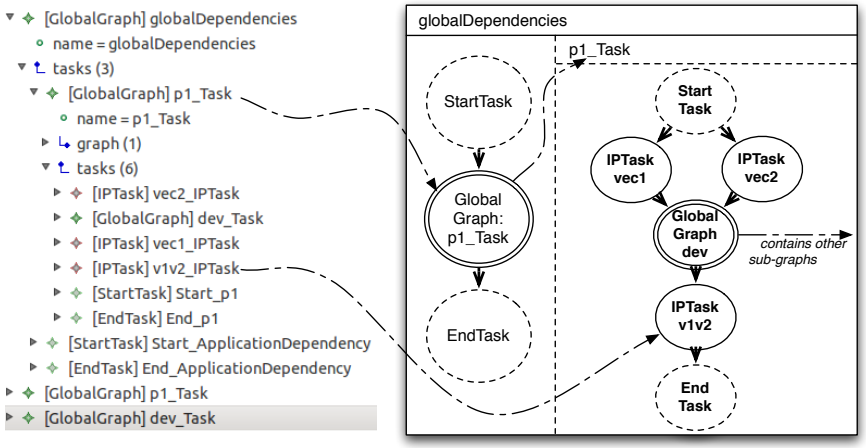


Figure 4.6: XMI Model Sample conforming for Local Task Metamodel and its Representation

for creating the graph representation as seen in the right side of the figure.

Once having the global task graph available in the model, we can propose a scheduling policy according to tasks and their processors where were allocated. The next subsection explains this policy.

4.2.2 *Choosing the Execution Order*

It is important to state that we were not searching an optimal scheduling policy at this level of task execution. For our target architecture, GPU, submitted tasks (kernels) run in the microprocessors according to the GPU scheduler and we do not have control on that. The aim is at creating a macro calling list for tasks globally defined in the model. Therefore, this ensures, at least, a coherence for exchanged data by the tasks. For independent tasks, however, OpenCL allows to define asynchronous task scheduling from host dispatcher. Thus, this avoids blocking sequential calls (*cf.* Subsection A.4.2).

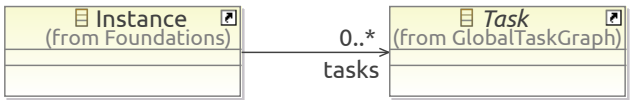


Figure 4.7: Scheduling Metamodel

The metamodel seen in Figure 4.7 is, in short, only a relationship between two external classes. An application instance contains the attribute *tasks* which is an ordered list of tasks from the global task graph. The process to define this list is explained in the Chapter 5. In advance, it is a simple graph scanning looking for the next available task that is not waiting for other task completing its work.

4.3 MEMORY MAPPING

We have proposed a metamodel whose objective is to make easier the variable declaration. The main problem resides in how to set a memory space, how to define allocations to different host and devices without forgetting the communication among them, and moreover, how to gather all data ports (*flowPorts*) defined in the application model that share the same memory address.

Figure 4.8 shows the metamodel that defines elements concerning to memory mapping. Two classes are created to add concepts for data allocation in a memory address space: *MemoryMap* and *DataAllocate*. An *AssemblyPart*, an instance of memory *HwRAM* component in the defined architecture model in this case, must have one *memoryMapping* of *MemoryMap* type which is composed or not of *dataAllocations* of the *DataAllocate* type. Each *dataAllocation* has its own data scope. This scope (*spaceAddress*) is defined by *addressSpaceQualifiers* = {*global*,

constant, local, private}, as specified in compute device memory hierarchy (*cf.* Section A.4.1) [1]. Nevertheless, this metamodel can be easily generalized to most hardware platforms.

1. Although the memory mapping metamodel intends to be a generic metamodel for several target systems, it is strongly inspired on GPU memory model. In fact, we find this memory model extensive enough to meet most systems.

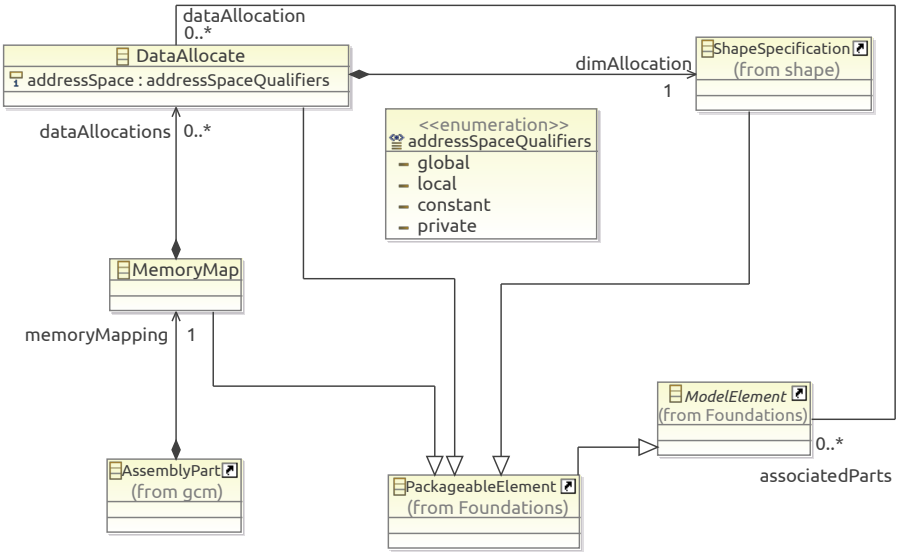


Figure 4.8: Memory Mapping Metamodel

As a result of application of this metamodel, Figure 4.9 presents an XML Metadata Interchange (XMI) model sample that allows us to generate source code for variable definition.

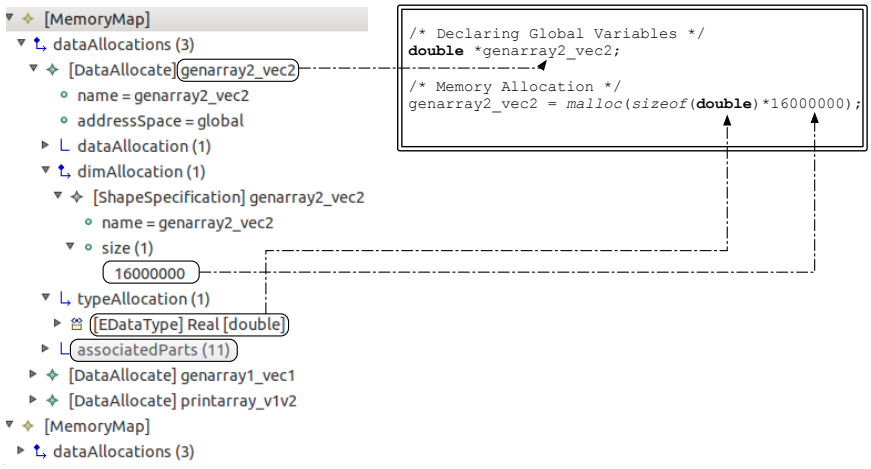


Figure 4.9: XMI Model Sample for Memory Mapping and an example of generated source code based on this model

The example illustrated in Figure 4.9 deals with the declaration and memory allocation of a variable in C language. The variable name was defined based on the *Data Allocate* name, the *addressSpace* attribute is used to define the scope. For the dynamic memory allocation, the type and the occupied size are obtained from *typeAllocation* and from the

The variable scope depends on the zone where it is declared in the program.

attribute *size* in *dimAllocation*, respectively. The *DataAllocate* brings yet an other attribute called *associatedParts* containing 12 elements. Here we list all *flowPorts* and derived elements which refer to this same memory space, using either temporal or spatial references.

4.4 HYBRID

The closest metamodel to OpenCL definitions and syntax is the Hybrid metamodel that allows us to generate the Hybrid model. Differently from the two earlier metamodels, this one is heavily attached to the target platform. We propose this metamodel as being the last one in our hierarchy of metamodels. It sums up all previous analyses aiming at creating a model with components that match with elements in a real program. Figure 4.10 depicts this metamodel which helps to create a real compilable application. This metamodel is strongly inspired by GPGPU programming model seen in languages such as OpenCL. Actually, even if OpenCL defines a generic compute device in its platform model, GPU devices are better suitable to this metamodel approach.

Analyzing the metamodel, we can verify that a new hierarchy of elements is added to the application instance in the affected model. The start element in this hierarchy is the *HybridApp*, the global representation of OpenCL application itself. From this point, we split the application into two parts: *HostSide* and *DeviceSide*. These parts are unique instances and they characterize technically the *host* and *compute device* roles defined by OpenCL. The *HostSide* and *DeviceSide* inherit from *ExecutionSide*. Thus, both can instantiate *Functions* that represent all possible tasks from application model. *Kernels*, a special function in the device, are tasks with only one *LaunchTopology*. Launching a kernel requires the specification of the dimension and size of the work-groups and work-items (*cf.* Section A.4.2). In order to define a dimension, we analyze three aspects: total number of work-items, data structure organization, and the shape of the task which is defined by the model designer. The LaunchTopology element specified in Figure 4.10 has 3 attributes (*dim, global, local*). The attribute *dim* is a 2-elements array containing the shape dimension of the respective kernel and total(shape internal product) of repetitions. The other attributes (*global* and *local*) are respectively the total number of work-items and the number of work-items per block or work-group. Details about how to define these values is seen in Chapter 5.

At the scheduling level, the metamodel provides an ordered list of functions (scheduling relationship). This list is associated to each composed task (hierarchical) according to function call order. Therefore, as kernels (from device side) and main function (from host side) are composed tasks, each one has an ordered list based on the scheduling previously defined in the model. Another relationship, the list (func-

tions), allows to enumerate other functions hierarchically as children of the function itself.

The Memory Allocation model is used as source to define variables in the Hybrid model. Indeed, characteristics such as scope, type, size, reading/writing are information easily retrieved from the precedent model of memory allocation. However, when we program in OpenCL, it is important take into account characteristics related to variables other than conventional ones. In order to cover every variable concern, we propose two relationship types between variables that help to implement the distributed aspect inherent to OpenCL memory model. The first one is *refersTo* relationship. This relationship allows to find which host variables must be transferred to device variables and vice-versa. The second one, composed by *indexin* and *indexout*, allows to express the tiler (*cf.* Subsection A.4.2.5) connector between *flowPort* elements. This information is a key aspect to determine which part of whole input data a work-item will process. In this case, tiler functions provide, for example, how each work-item gathers or scatters data from/into global memory.

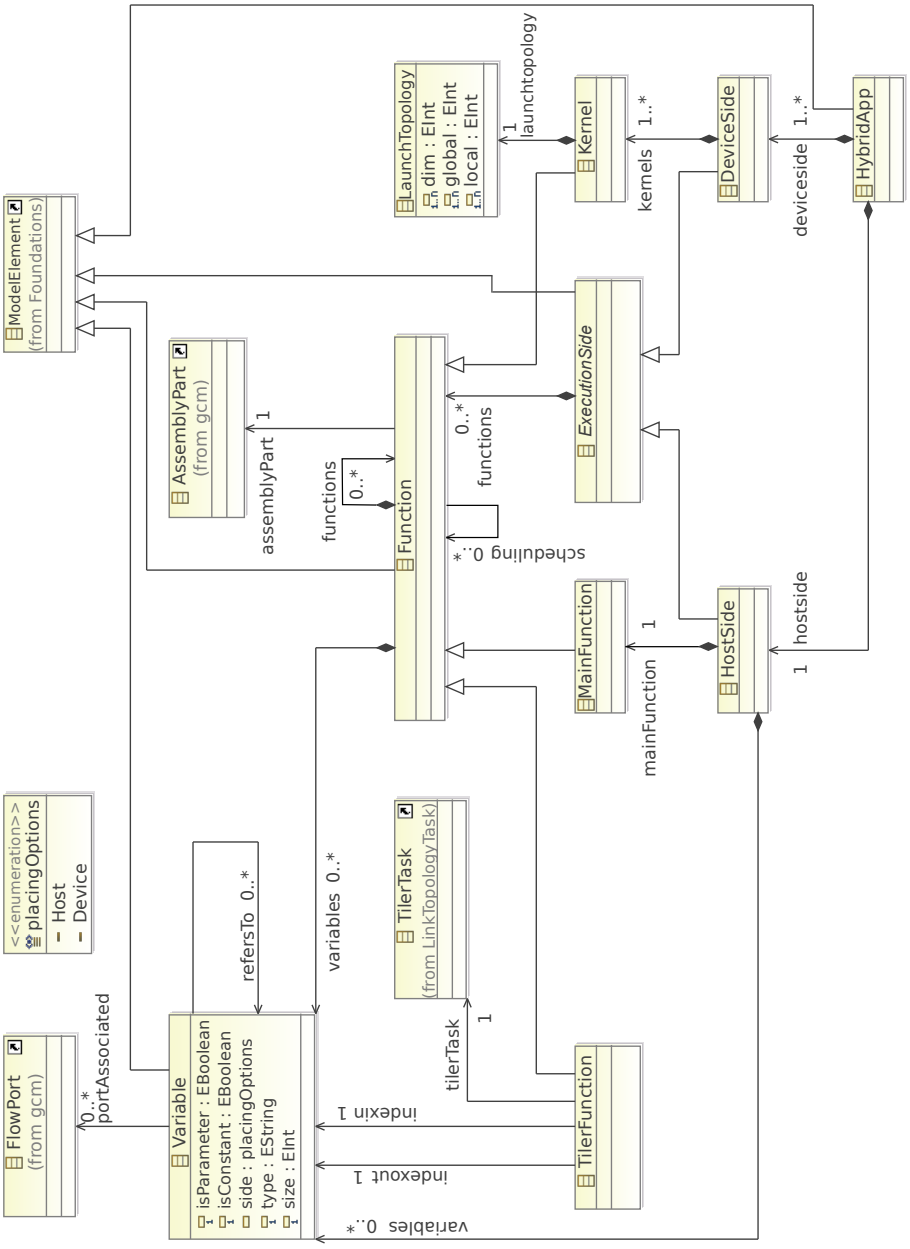


Figure 4.10: Hybrid Metamodel

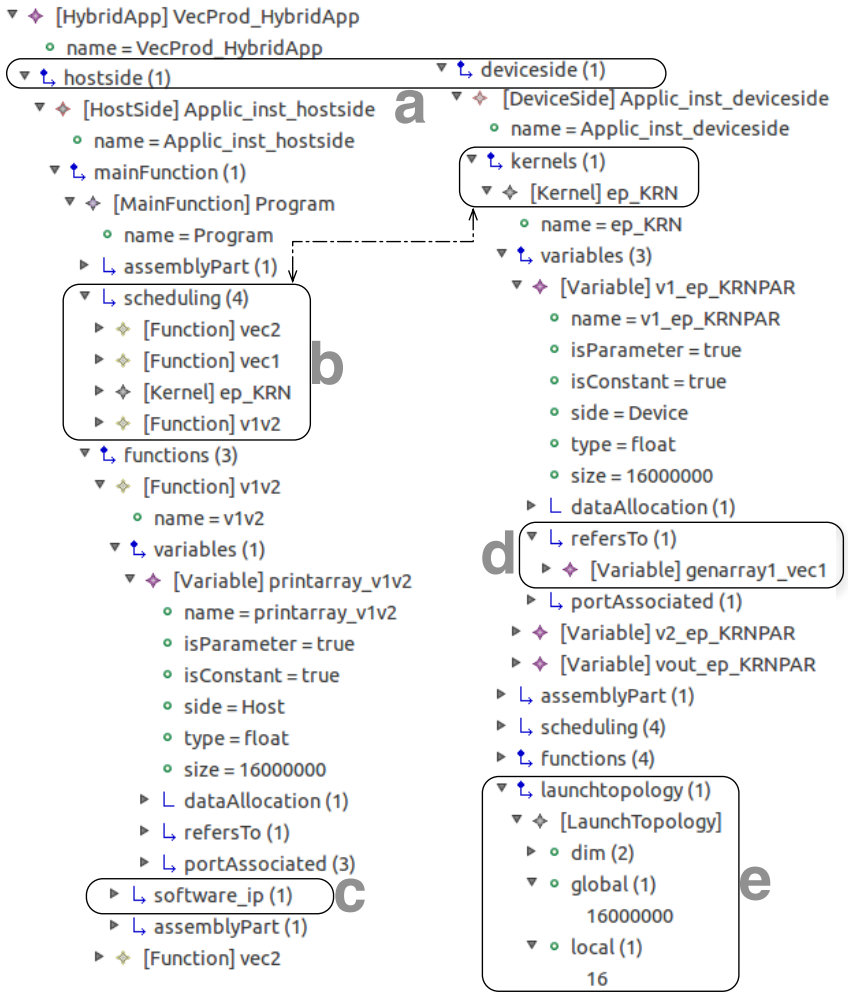


Figure 4.11: XMI Model Sample for Hybrid Application

To clarify this metamodel usage, Figure 4.11 shows a model sample based on the Hybrid metamodel. Some highlighted points in the figure illustrate the main applications of this metamodel approach.

(a) By default, a Hybrid Application is composed of two parts: one *hostside* and one or more *deviceside*. Here we have one *Applic_inst_hostside* and one *Applic_inst_deviceside*.

(b) The previous scheduling policy defined generically in the scheduling metamodel is adapted to our function list. Notice here that we have 4 functions as an ordered list: {*vec2*,*vec1*,*ep_KRN*,*v1v2*}. Moreover, the *kernel function* is seen as a function call by the host.

(c) The deployment is indicated in this point. The whole structure involving code snippet or, in some cases, the binary, parameters and its order, and any additional auxiliary files are associated here to the depoloyment model.

(d) This variable attribute allows us to make one of most important aspects in OpenCL programming: data transfers between host and device. In this example, the variable *v1_ep_KRNPAR* refers to the variable *genarray1_vec1*. During execution, whenever we need to send or receive data by host and devices, this attribute will be taken into account.

(e) The launch topology is indicated in this relationship. As an example, globally we have 16 millions of work-items distributed into work-groups of 16 work-items locally. Thus, we have 1 million work-groups composed of 16 work-items.

4.5 CONCLUSION

In this chapter, we presented three key metamodels that comprehend three global concerns for creating an whole application: scheduling policy, memory handling, and the target specificities. Although the high level model designing aims to create parallel applications, it remains at a high abstraction level and lacks or does not provides explicitly important information when we aim a specific target. The first two metamodels presented here can be easily used in many other target platforms. Scheduling and Memory Allocation are generic needs of most applications for any architecture. Nevertheless, the Hybrid metamodel was developed to attain our code generation approach and its understanding enlightens most aspects of our whole approach itself.

5

MODELS TOWARDS CODE

Chapter Contents

The core of our code generation approach lies mainly in the model transformations. We have defined several model transformations modules which now, along with other ones, are part of the Gaspard2 Model Transformation Library whose structure is shown in Figure 5.1, already introduced in Chapter 2. Choosing the suitable transformations modules is part of the compiling engineering process. As an MDE approach, the new branch proposed for Gaspard2 comprehends all models, metamodels, transformation modules, and, foremost, how to determine the compiling process layers in order to achieve all necessary model element analysis. In this chapter, we present our model transformation chain and how it works regarding the metamodels previously depicted in Chapter 4 as well as metamodels introduced in this chapter.

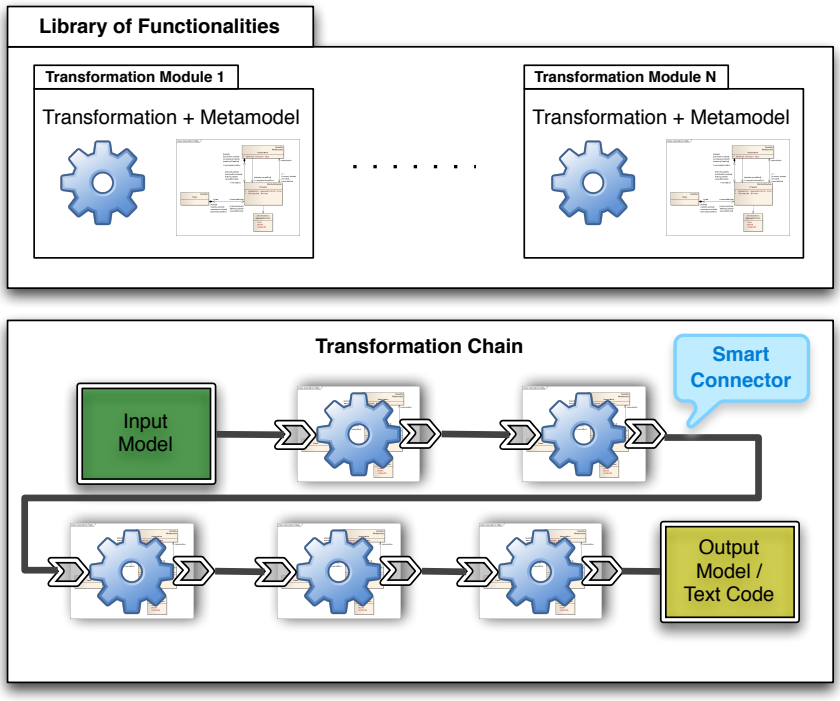


Figure 5.1: Gaspard2 Library of Functionalities and Chaining Process

5.1 BUILDING A TRANSFORMATION MODULE

Our transformations modules are unidirectional, hence, they have only one mode of execution: i. e., they always take the same type of input and produce the same type of output. Unidirectional model transformations are useful in compilation-like situations, where any produced output model is then read-only to next transformations. To implement our transformations, we have used the OMG-standardized model transformation language known as Query/View/Transformation Operational (QVTO) (*cf.* Section B.3).

In order to facilitate the transformation chaining engine (discussed in next section), we used the transformation scheme seen in Figure 5.2. For each transformation module, we have only one input model and only one output model. The reason for this is simple: we aimed to keep the transformations the least complex possible, with fewer rules and model affectations. However, we kept in mind the design of transformations of full functionalities, e. g. the scheduling or memory allocations. This avoids monolithic transformations and provides a layered compilation scheme.

Back to Figure 5.2, "Model A" conforms to "Metamodel A" is the input model for our transformation module example. However, as we copy all elements from the input model into the output model for future analysis within other transformations, we have decided

to create *inout* transformations. It means that our input model is the same output model and both are conforming to "Metamodel B". Actually, the "Metamodel B" is the "Metamodel A" plus "Metamodel B Delta". Those operations running on the metamodels are more than a simple adding. Sometimes, it is necessary to change relationships or attributes types, for instance. In this case, no new element was added, but it just had its relationship or attribute modified.

In order to adapt the "Model A" to "Model B", used as inout model, the Gaspard2 team proposed the MD Factory tool allowing transparently to do this adaptation when we chain two transformation modules. This is implemented by that we call "Smart Connector" seen in Figure 5.1.

In summary, the process of building a transformation module takes into account that every input model **A** must be converted (step 1 in Figure 5.2) into another model **B** which is consistent with the model **A**, but it is conforming to metamodel **B**. The metamodel **B** is the result of the fusion of metamodel **A**, whose model **A** conforms, and the metamodel **B Delta** that contains the necessary modifications which will be taken into account by the transformation rules. The transformation takes (step 2) the model **B** and transforms it into the model **B'** (step 3).

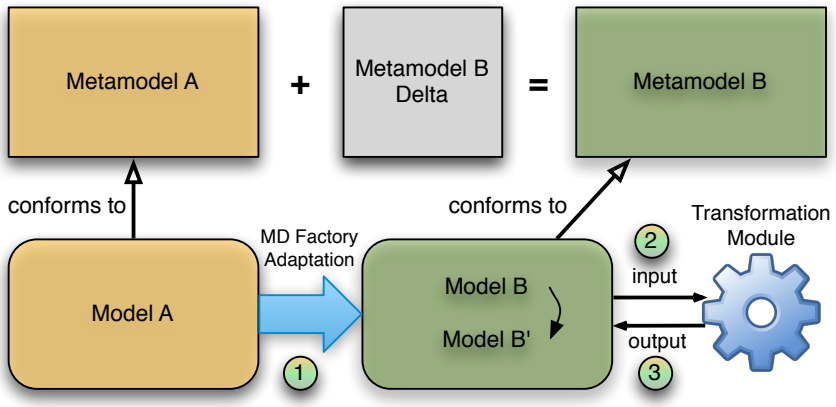


Figure 5.2: Model Transformation Scheme used in Gaspard2

5.2 CHAINING MODEL TRANSFORMATIONS

Chaining transformations means to put them into a sequentially ordered list that takes into account their inter-dependencies. Figure 5.3 presents the transformation chain for our approach. Except for the modules 8 and 9, the models do not aim any specific target platform. Indeed, they were designed to be the most generic possible. Thus, this guarantees the reuse feature for these modules. Some of the

chosen transformation modules could run in parallel once they do not have model dependency. However, for the time being, we did not implement transformation chains as graphs. In fact, this has no high impact on model compiling performance, thus, transformations are chained sequentially.

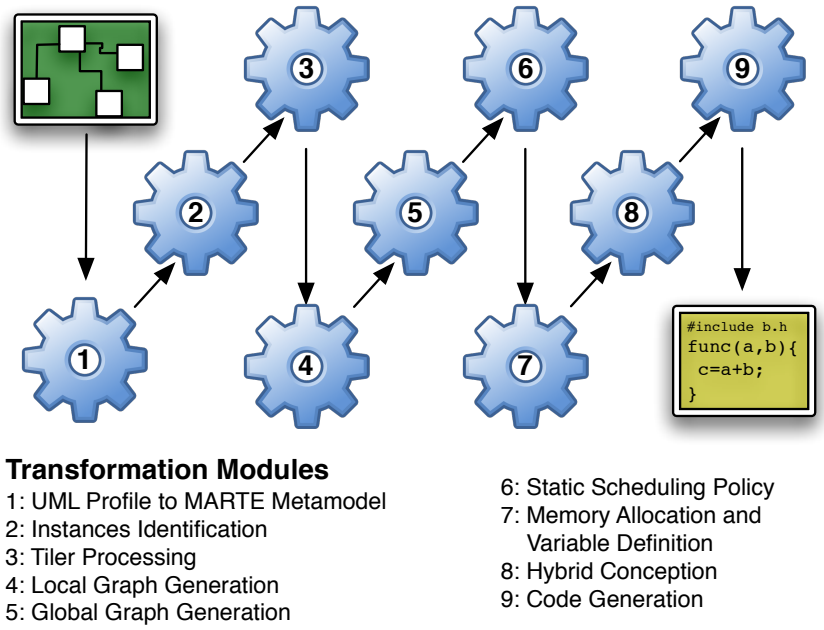


Figure 5.3: The UML/MARTE-to-OpenCL Transformation Chain

5.3 GENERIC TRANSFORMATION MODULES

Some of transformation modules were already available in Gaspard2 library to methodology providers wanting to use them for some particular purpose. However, besides the new proposed transformation modules, our findings affected all modules within the chain. For this reason, and also for a better overall understanding, we present in the next subsections all involved modules for the code generation of this particular target, OpenCL.

5.3.1 *UML Profile to MARTE Metamodel (1)*

Although this transformation module's name contains "profile to metamodel", it does not produce a metamodel as an output. Earlier, in Chapter 2, we explained that the choice for UML was due to its broad application domain, its large available tools, and mainly, its MARTE profile as standard for *real-time* systems and support for description

of parallelism. However, during the developing of the transformation modules, we realized that converting the UML profile for MARTE into a metamodel for MARTE would make the transformation job easier. In fact, this process means just to adapt a model conforming to UML into a model conforming to MARTE metamodel. Having this model, the remaining transformations do not need to deal with all unnecessary extra complexity of UML.

Listing 5.1: QVTO snippet from UML to MARTE Metamodel Transformation

```
1 mapping UML::Class::class2HW_Resource() : HRM::HW_Resource
2 when{
3  not self.getAppliedStereotype(source.MARTE_STEREOTYPE_HWRESOURCE).
       oclIsUndefined()
4 }
5 {
6  init{
7   var stereotype := self.getAppliedStereotype(source.
       MARTE_STEREOTYPE_HWRESOURCE);
8  }
9  description:= object nfps_types::NFP_String{
10   value := self.getValue(stereotype,'description').oclAsType(String);
11  }
12 }
```

Listing 5.1 shows a snippet of QVTO-language source code for this transformation module. Obviously we do not intend to explain all the source code (about 1700 lines), but showing this mapping rule can illustrate how the other rules were implemented. In line 1, we declare a special mapping rule called `class2HW_Resource`, this rule is applied to UML classes and produces, as output, an element conforming to HW_Resource defined in the Hardware Resources Modeling (HRM) package from the MARTE metamodel (*cf.* Figure B.3). The *when* section in lines 2-4 ensures that this mapping rule will be only applied on classes with *HW_Resource* stereotype. Finally, lines 6-12 consist in setting the value of the *description* attribute to the related value defined in the stereotype from MARTE profile.

5.3.2 *Instances Identification (2)*

In UML 2.0, ports represent an interaction point between a component and its environment and its internal parts. Nevertheless, parts of components whose type is a component owning ports do not have instances of those ports. The component hierarchy proposed in Gaspard2 relies on composed components whose parts must be interconnected by their ports to assure the data flow. In order to add the instance of ports for each part within the component, Gaspard2 proposes the transformation module "Instances Idenfitication". In Figure 5.4, we present the general view of the metamodel associated to this transformation. It is strongly inspired on MARTE profile. However, it adds three new

classes *AbstractPart*, *InteractionPort*, and *PortPart* that extend MARTE to allow us to create port instances.

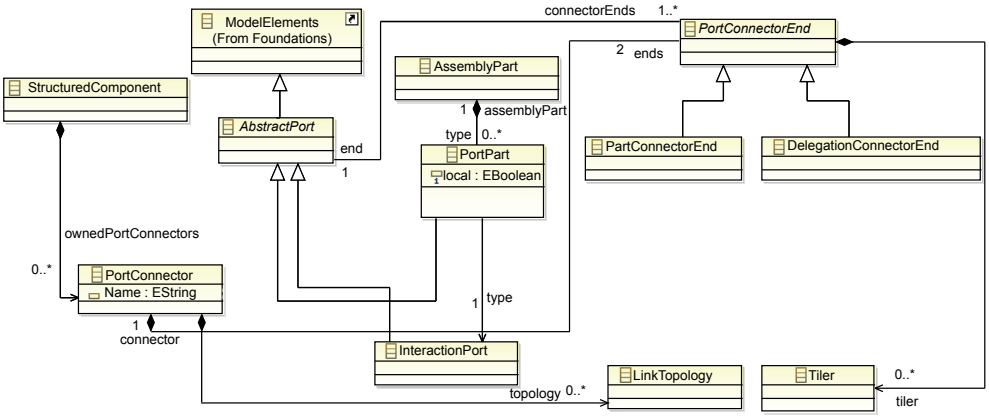


Figure 5.4: Instances Identification Metamodel

5.3.3 *Tiler Processing (3)*

Tilers are part of ARRAYOL specification language. Appendix B presents an introduction to ARRAYOL in MARTE context. The model designer specifies *tilers* as stereotypes applied directly to port connectors. However, if we consider *tilers* as data scattering and gathering operations, it is not clear who (processing element) executes these operations. Hence, this module transforms every *tiler* specified on higher level models to *tiler tasks* as illustrated in Figure 5.5. Later, these newly *tiler tasks* are undertaken by the same processor element of corresponding tasks previously connected to those *tilers*.

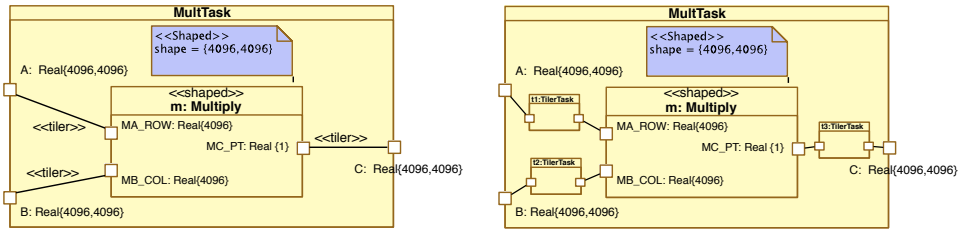


Figure 5.5: Transforming Tiler Connectors to Tiler Tasks

5.3.4 *Task Graph and Scheduling (4,5,6)*

The transformation modules 4,5, and 6 in Figure 5.1 are responsible for creating task graphs and a trivial scheduling policy. They are based on the metamodels described in earlier chapter. A first transformation captures the tasks for each allocated processor element. Then, it

creates a local Directed Acyclic Graph (DAG) according to their data-dependency. Afterwards, another unique DAG is generated by a second transformation that regards the data-dependencies under the global point of view.

The last transformation in this set defines a static scheduling by analyzing the DAG seeking any valid execution path. Due to multiplicity of valid paths, this transformation can generate random valid lists.

Although there are many algorithms [93, 145] that provide a better DAG analysis, this is our scheduling solution for our approach. In fact, we do not need anything more complex because this scheduling deals only with the sequential part of our generated code. The parallel part is entirely managed by the scheduling controller of the GPU.

5.4 MEMORY ALLOCATION AND VARIABLE DEFINITIONS (7)

From this section we start presenting an in-depth analysis of transformation modules. In fact, this and the following transformation were entirely designed during the works of our methodology approach, even if the memory mapping was thought to be a generic transformation to many other target platforms and transformation chains.

Figure 5.6 shows the overall structure of the memory mapping transformation module. It uses the memory mapping metamodel presented in previous chapter. Here, we explain the main phases of this transformation by enumerating the key rules as follow:

1. **addMemoryMap**: this rule seeks all «*HwRAM*» elements in the model and creates a child element *memorymap* that will contain the *data allocations* (**defineBasicDataAllocations**). The basic data allocations are created from the allocated *flowports* of the model.
2. **defineScope**: by analyzing the «*HwRAM*» element in the memory hierarchy, this rule allows for defining the scope, e. g. "global", of a data allocation.
3. **propagateDataAllocation**: once are defined the data allocations, this rule is able to propagate this allocation to every element referring to the same allocation. For instance, a not-allocated *flowport* shares the same allocation of a directly connected *flowport* that have already an allocation. This situation can be observed in Figure 5.8 for the output port of the task *magen* and the first input port of the task *mtask*.
4. **createTilerTaskDA**: in Figure 5.5, 3 *tiler tasks* are created in *tiler processing* transformation module. This rule propagates existing data allocations to *tiler tasks* ports.
5. **createVirtualIPSoftIPDA**: the matrix multiplication example in Chapter 3 shows the deployment phase when we define two new elements in the model: the *virtual IP* and the *software IP*. Like

their corresponding task components, these two new elements have *flowports* that need a data allocation reference. However, in this case, no variable reference those ports, it exists only to ensure the link between IP parameters and the data allocation reference in the memory map.

x. **createAffectationDataAllocation**: a special rule allows to create and associate special data allocations for auxiliary variables. In our approach, this kind of variable is handled by static allocations. Therefore, this rule does not have any effect on our code generation process. It exists to retain the compatibility for other branches, such as Pthread.

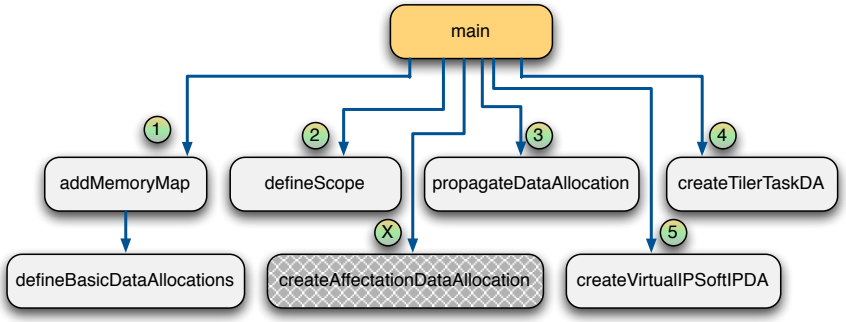


Figure 5.6: Memory Mapping Transformation Module Overview

At the end, this transformation creates new elements in the model. These elements summarize all data and the relationship among tasks concerning data communication.

5.5 HYBRID CONCEPTION (8)

Despite other important transformations, this transformation can be considered the most important one. Indeed, this transformation summarizes all explicitly modeled or implicitly defined elements by earlier transformations into a single structure. Moreover, as seen in Chapter 4, the metamodel used to generate the result model defines elements that have a terminology closer to our target platform. It means the distinction between host and device, kernels, and so on.

In this section, we present which model elements were used to construct each part of the result structure, as well as the reason of this choice and mapping rules strategy.

5.5.1 *General Structure*

Figure 5.7 presents an overview of this transformation module. At first, the *main* function starts the creation of a hybrid application call-

ing the mapping rule `createHybridApp`. This produces the *HybridApp* element as a child of the general instance of the application. From this point, we split the structure into two *sides*, even though we keep some connection points between both *sides* to provide information about relationship between elements. Next subsections emphasize how the remaining rules work.

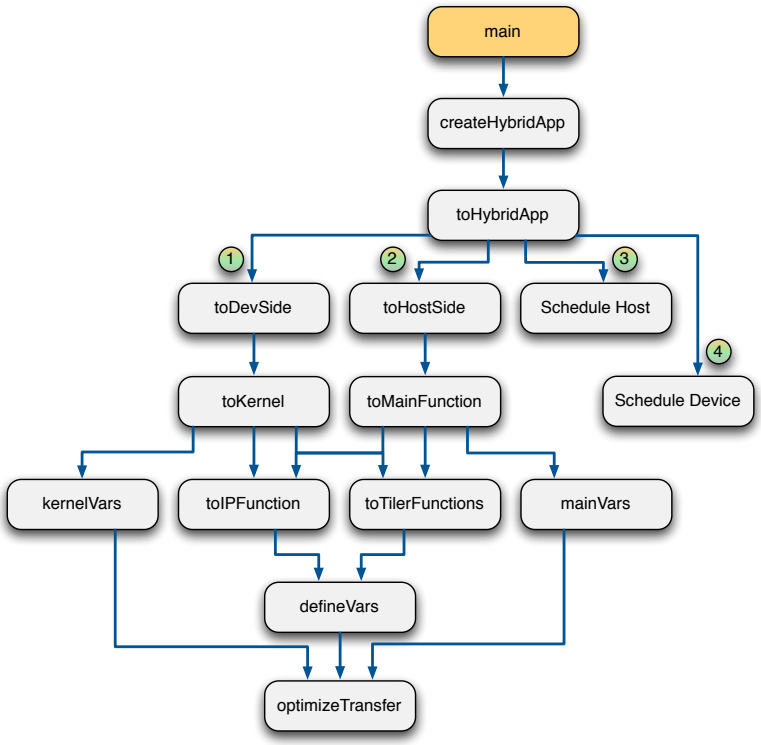


Figure 5.7: Hybrid Conception Transformation Module Overview

5.5.2 *Identifying Kernels*

On the device side, a transformation rule starts creating the kernels of the application (`toKernel` rule). Kernels are identified by tasks allocated onto devices as exemplified in Figure 5.8. Here, *abstraction* connectors stereotyped as *«allocate»* from MARTE and the hardware resource can be filtered by checking its description like in the operation: **`description.value->any(true)="Host"`**.

The generated kernel has been assigned, at this moment, its *name* and its *launchtopology*[16] . Regarding the topology, we have tested two solutions to its specification. In general, this does not impact directly on the code's functionality, however this can strongly impact on performance when we are looking for the optimal processor occupancy (*cf.* Chapter 6). Listing 5.2 presents the QVTO code for the computation of the kernel launch topology. The algorithm behind this listing aims to

[16] *For us, the launch topology of a kernel relies on its grid geometry. When a kernel is launched we have to define how many work-items and work-groups will compose its grid. Usually this grid have up to 3 dimensions for work-items and for work-groups.*

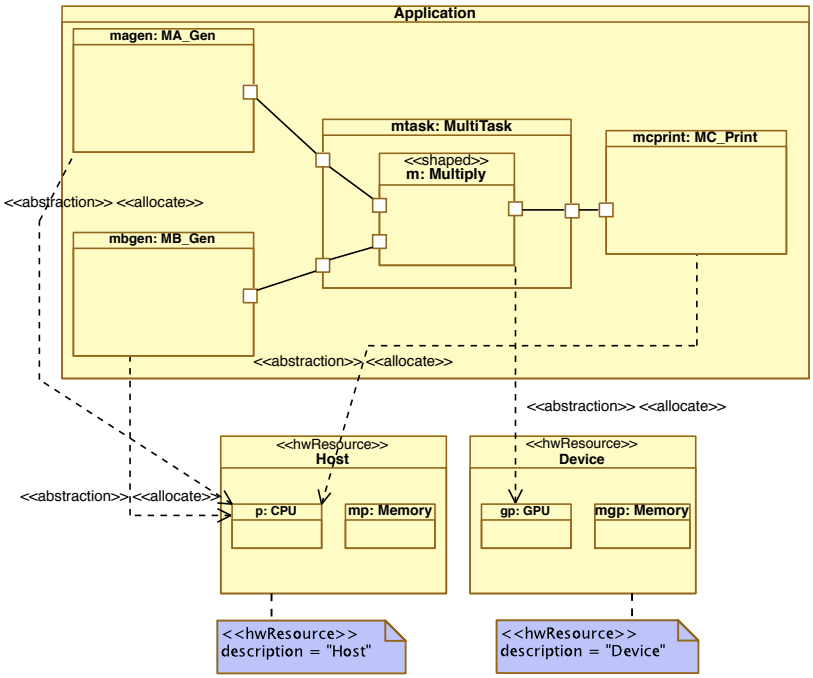


Figure 5.8: Distinct task allocation onto available processors

provide groups of n work-items, where n respect (being a multiple) typical *warp* specifications of GPU devices . After some tests and due to dependency of this information with the modeled application, we have decided to change this strategy to a simpler one as seen in Listing 5.3. Here, the launch topology is obtained directly from the tasks' shape. Hence, model designers can decide the kernel topology by themselves and regarding application's constraints. Nevertheless, in order to provide resources to analyze their decisions, we proposed an optimization technique based on profiling results to improve processor occupancy according to kernel topology. This technique is presented in Chapter 6.

5.5.3 *Functions and Variables*

The remaining composition of a kernel is defined by the next rules: `kernelVars` uses *flowPorts* and elements from the Memory Allocation transformation (7) results, which belong to the original task, to define the kernel parameters; `toIPFunction` creates every function regarding the IP calls from elementary tasks in the model and that are part of the task used to create the kernel; `toTilerFunction` produces special functions based on *tilers* tasks produced in Tiler Processing transformation (*cf.* Subsection 5.3.3). In OpenCL, in general each work-item gathers its data based on its indexes directly from the device global memory. Therefore, it is coherent to put the *tilers* within kernels as sub-tasks

along with IP tasks and, thus, they are part of the scheduling list of the kernel.

Once we have functions from IPs and *tilers*, we can define their variables by calling the rule `defineVars`. Again, the rule looks for data from *flowPorts* and memory allocation model to create variables associated to functions. The parameter order relies on the deployment model which describes the IP behavior.

5.5.4 *The Main Function*

Similarly to device side, on the host side, we have to create special functions (starting in the rule identified by the number 2 in Figure 5.7). However, differently from kernels, on the host side we have only one initial function called main[17] created by the `toMainFunction` rule. For this special function, like for the kernels, we call the rules for creation the IP and *tiler* functions, as well as the global variables produced by the `mainVars` rule (similar to `kernelVars`).

[17] *In many programming languages, the main function is where a program starts execution. Thus, we keep rather this terminology than another more generic concept.*

5.5.5 *The Relationship among Variables*

In order to establish the link between host and device for data transfers, the rule *optimizeTransfer* operates on previously defined variables. Here, the aim is to define which variables have co-variables in the opposite side. This is an important rule because the most part of the code optimization lies in memory transfers. In order to achieve this goal, the algorithm of this rule takes into account model analysis looking for farther tasks which make use of these variables. The aim consists in avoiding extra transfers as exemplified in Chapter 6 which deals with optimization of the code.

5.5.6 *Summarizing the Scheduling*

The numbers 3 and 4 in Figure 5.7 deal with scheduling. They are the last rules in the transformation because they need all the functions (this includes functions from IPs and *tilers*) previously generated for host and device. After running these rules, we have sequential lists of functions. Again, a link between host and device is taken into account in this rule. Actually, kernel functions are present in the scheduling list of the host. All kernel launches are managed by hosts, so in order to assure the kernel call at the right moment we have inserted kernel functions within host's list. Figure 5.9 illustrates the result of this process.

Listing 5.2: launchtopology computation

```
1 ...//topology x,y,z
2 launchtopology :=  object hybridapi::HYBRIDAPI::LaunchTopology{
```

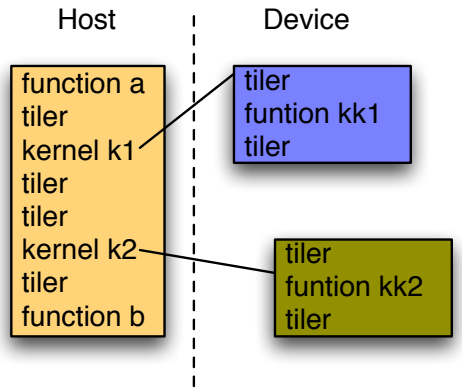


Figure 5.9: Scheduling lists and their interconnections.

```
3          dim := dims.size->size();
4          dim += dims.shapeprod();
5          var i : Integer = 1;
6          var j : Integer = dim->at(1);
7          while(i<=j) {
8             dim += dims.size->at(i);
9             if(dims.size->at(i)->notEmpty()) then {
10              switch {
11                case (j=3) local+=8;
12                case (j=2) local+=16;
13                case (j=1) local:=32;
14              };
15             global+=(dims.size->at(i)/local->at(i) + 0.49999999).
                   round()*(local->at(i));
16            } endif;
17            i:=i+1;
18          }
19       };
```

Listing 5.3: launchtopology computation directly from task's shape

```
1 .../topology x,y,z directly from the shape
2 launchtopology :=  object hybridapi::HYBRIDAPI::LaunchTopology{
3                       dim := 1;
4        dim += dims.shapeprod();
5                       local := dims.size->at(1);
6                       global := dims.shapeprod();
7     };
8 ...
```

5.6 CODE GENERATION (9)

This transformation maps an abstract model to text, i. e. the source code. At this point, we are already able to write out the earlier

analyzed elements directly to code. We have considered that the source code for an OpenCL application is composed of:

1. makefile;
2. `.cl` files, one for each kernel;
3. `.cpp` file, one for the whole application;
4. `.h` header files;

The directory and makefile structures are based on NVidia's API. However, small modifications can adapt the result code source to several vendor's API. In this subsection, we present our strategy to generate the code files using Acceleo. This code generation language uses a template based approach. With this approach, a template is a text containing dedicated part where the text will be computed from elements provided by the inputs models. We have created one dedicated template for each type of the files structure (makefile, .cl, .cpp, .h).

5.6.1 *Creating the makefile and header files*

For the makefile, the transformation provides file names and possible links to libraries and compilation directives. That information is gathered from the deployment model and depends on the provided IPs.

Listing 5.4: CL Code Template

```
1  <%
2  metamodel /fr.inria.dart.gaspard2.metamodel.hybridapi/hybridapi.ecore
3  %>
4  <%script type="HybridApp" name="Makefile" file="generated/Makefile"%>
5  ####################################################
6  # Tip: Copy these files on src folder in NVIDIA_GPU_COMPUTING
7  EXECUTABLE  := ocl<%name%>
8  # C/C++ source files (compiled with gcc / c++)
9  CCFILES   := ocl<%name%>.cpp
10 LIBUSRLNK := <%hostside.functions.software_ip.codeFile.linkDirective.
       trim().sep(" ")%>
11
12 include ../../common/common_opencl.mk
```

The name and path of the header files are obtained from the model, then they are copied to the same directory where the code is generated.

5.6.2 *Creating OpenCL Kernels Files*

All header and IP files are provided as files available in the code generation environment. Figure 5.10 shows two samples of this kind of files available for the matrix multiplication application modeling. The function declared in the IP, for instance, operates on single elements of

data. The parallelism and distribution and communication of data are entirely handled by the generated code from the application model.

matrix.h

```
#define MAX_N 4096
```

multinstance_IP.cl

```
/* Gaspard2 MDE
   IP name: MultInstance
   function name: mult
   Parameters: a,b,c */
void mult(
  const        float* a,
  const        float* b,
               float* c)
{
      c[0]=a[0]*b[0];
}
```

Figure 5.10: Samples of IP and header files for the matrix multiplication application

[18] *The complete template has about 200 lines.*

Analyzing a small part[18] of the template for `.cl` files in Listing 5.5, we can emphasize the IP functions inclusion in lines 17,37-41. The script gets the file name and path directly from the model *HybridAPI* of the functions related to the analyzed kernel. Then, all variables (kernel and function parameters) are declared as seen in lines 22,24-26. For the ordered function call list, the instruction in line 33 prepares the IP and *tiler* calls.

Listing 5.5: CL Code Template

```
55 ...
56 //__________________IP Functions_____________________
57 <%functions.ip%>
58
59 //_________________________________________________
60 //Kernel Start
61 __kernel void <%name%>(uint   iNumElements,
62     <%variables.declaration.sep(",")%>)
63 {
64     <%if (functions.variables.nSize()>0){%>
65         <%functions.variables.declarationInFct.sep(";")%>;
66     <%}%>
67     //get index into global data array  (x,y,z) x + sx*y + (sx*sy)*z
68     int iGID = get_global_id(0) + get_global_size(0)*get_global_id(1)
          + get_global_size(0)*get_global_size(1)*get_global_id(2);
69
70     // bound check
71     if (iGID < iNumElements)
72     {
73     <%scheduling.schedule.sep("\n")%> //IP and Tiler functions
74     } else return;
75 }
76 ...
77 <%script type="HybridAPI.Function" name="ip"%>
78 //########<%name%>: file: <%software_ip.filePath%>##########
79  <%
```

```
80  getFileContent("fr.inria.dart.gaspard2.codegeneration.gpu_opencl/src
        /IP/"+software_ip.filePath)
81  %>
82 <%script type="HybridAPI.Function" name="schedule"%>
83 <%software_ip.entryName%>(<%for (variables) {%><%if (i()!=0){%>, <%}
       %><%name%><%}%>);
84 <%script type="HybridAPI.TilerFunction" name="schedule"%>
85 <%self.calculate%>
86 ...
```

5.6.3 *Creating C/C++ Files*

The process of creating the `.cpp` files is similar to `.cl` one. However, on the host side, we have a few extra operations. On the host side, it is necessary to take into account: *device detection, memory allocation for variables, kernel load and compilation, parameter definition,* and *memory transfers*. Those operations are detailed in Chapter A. However, in order to clarify our strategy to some of the operations, we emphasize three important ones: launch topology definition, memory transfers, and argument setup for kernel launching.

To define the grid of work-items for the kernel launch context, we use the element *launchtopology* created in the *Hybrid* transformation as seen in the previous section. To understand this process we analyze the Listing 5.6. In lines 102,103, we define how many dimensions for work-items group and work-groups group. In order to avoid work-items working outside allocate data, in line 103, we define the total number of work-items will really operate on the data. The remaining code deals with enough memory space allocation for variables which will define the grid parameters and the last part defines their values. As an example, Figure 5.11 shows graphically a possible configuration.[19]

Listing 5.6: Launch Topology to Grid definition

```
445 ...
446 //threads environment setup
447 Ndimwi  = <%launchtopology.dim.nGet(0)%>;  //WI Dimension
448 Ndimwg  = <%launchtopology.dim.nGet(1)%>;  //WG Dimension
449 Nelem = <%launchtopology.dim.nGet(2)%>;  //Boundery
450 free(szLocalWorkSize);
451 free(szGlobalWorkSize);
452 szLocalWorkSize = (size_t *)malloc(sizeof(size_t) *Ndimwi);
453 szGlobalWorkSize = (size_t *)malloc(sizeof(size_t) *Ndimwg);
454
455 <%for (launchtopology.local){%>
456 szLocalWorkSize[<%i()%>] = <%self%>;
457 <%}%>
458 <%for (launchtopology.global){%>
459 szGlobalWorkSize[<%i()%>] = <%self%>;
460 <%}%>
461 ...
```

[19] *OpenCL defines that the global size uses work-items as unit and not work-groups. Sometimes, in order to fit the same number of work-items into every work-group, we use some round up function. In this case, the boundary verification is strongly necessary.*

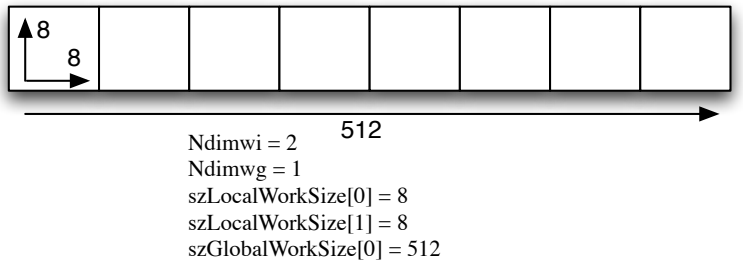


Figure 5.11: Grid Example

Regarding the memory transfers, to write out instructions responsible for the data copies, we have chosen the functions `clEnqueueReadBuffer` and `clEnqueueWriteBuffer`. Besides those, we can use `clCreateBuffer` during the memory allocation. However, we prefer the former ones because we consider that all allocations are already done and the copy actions can be made during kernel launches, wherever they were allocated. In the next chapter, we explain some memory transfer optimizations and we address those functions again.

Listing 5.7 presents the template segment responsible for the situation depicted in Figure 5.12. In line 482, we check the output (not constant) variables in kernel having a reference in the host side, such as "1" in the figure. In line 483, we define the copy instruction by providing type, size, and the name of the involved variables, "1" towards "2" in this case. The second part of the code deals with further transfers. A variable having two or more *refersTo* elements must have its data copied into referenced input(constant) variables, such as "2" to "4". An in-depth analysis of the optimization of this process is presented in Chapter 6 regarding transfers suppression.

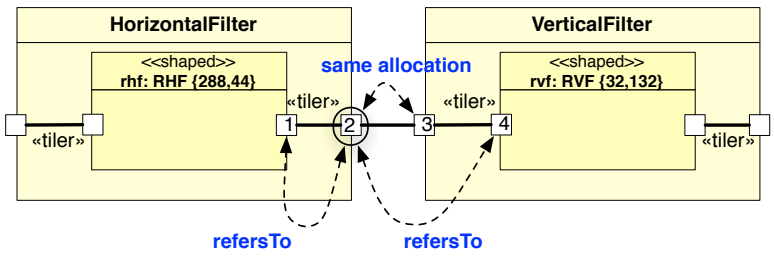


Figure 5.12: References for Memory Transfers

Listing 5.7: Memory Transfers

```
480 ...
481   <%for (variables){%>
482     <%if (!isConstant && refersTo.side=="Host"){%>
```

```
483     dartErr1 = clEnqueueReadBuffer(cqCommandQue, cmDev_<%name%>,
            CL_TRUE, 0, sizeof(cl_<%type%>)*<%size%>, <%refersTo.name%>,
            0, NULL, NULL);
484 <%-- if it has more than 1 refersTo, it is also an input var --%>
485       <%for (refersTo.refersTo){%>
486         <%if (isConstant){%>
487         dartErr1 = clEnqueueWriteBuffer(cqCommandQue, cmDev_<%name%>,
                CL_TRUE, 0, sizeof(cl_<%type%>)*<%size%>, <%refersTo.
                name%>, 0, NULL, NULL);
488         <%}%>
489       <%}%>
490     <%}%>
491   <%}%>
492 ...
```

Some applications require defining the launch topology with work-items number above the truly necessary. Indeed, if we seek performance, we have to take into account better grid definitions in order to assure better processor occupancy. For this reason, we always send the total number of work-items, which will really work on data, as argument. In Listing 5.8, we present the template for the definition of kernel arguments. In OpenCL, it is proposed a special function to declare the arguments and their order, `clSetKernelArg`. In line 461, *Nelem* is always the first argument (the "o" in the second parameter). Then, we save the respective kernel object in the stack (line 462) for future reference and we scan its variables (line 463). In line 464, we define the arguments. The order is random. However, this order is consistent with the order defined in the `.cl` file of the corresponding kernel.

We have masked some information about optimization in this chapter because we present it in the next one.

Listing 5.8: Setting Kernel Arguments

```
460 ...
461     dartErr1 = clSetKernelArg(ckKernel_<%name%>, 0, sizeof(cl_int), (
            void*)&Nelem);
462     <%self.push()%> <%-- kernel object in stack --%>
463     <%for (variables) {%>
464     dartErr1 |= clSetKernelArg(ckKernel_<%peek().name%>, <%i()+1%>,
            sizeof(cl_mem),  (void*)&cmDev_<%name%>);
465     <%}%>
466     <%self.pop()%> <%-- kernel object out of stack --%>
467 ...
```

5.6.4 *Extending the number of available devices*

Many GPGPU systems start to have multiple GPU devices to increase two constraints: processor resource and memory resource. While vendors release multi-GPU systems, programmers have to manually take into account the load balance and data communication among the available devices. Usually, this process is handled on the host side. In addition, converting an application written for a single GPU

to run on multiple devices generally requires rewriting the code, and sometimes fairly extensive modiÞcations are required. Ensuring consistency between the multiple copies of data makes this process more difficult for the programmer.

For the model designer, who does not necessarily have the knowledge about the device programming level, we propose a workload distribution based on task's shape and devices's shape. Indeed, for some constrained tasks, this distribution process is easier to compilers than to programmers. Actually, for each device, the programmer must take into account all procedures to launch a kernel. This means context creation, parameters settings, data transfers (host to device and vice versa), grid settings, so on. This is arduous for manually written code, but is fairly automatized process for model compilers.

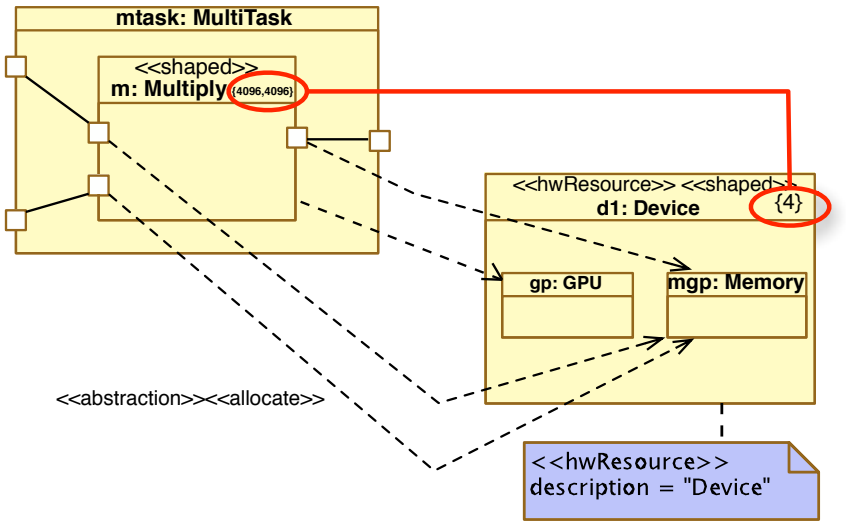


Figure 5.13: Multi-GPU Example

The process described here is part of the code generation phase and does not require complex changes in the high level specification model. Figure 5.13 is based on the Matrix Multiplication example presented in Chapter 3 and emphasizes the *shapes* of the task *m: Multuply* and the device for which this task is allocated. The task has *shape* value equals $\{4096, 4096\}$. Usually, in a mono device environment, it becomes 4096 work-groups with 4096 work-items each one. Expanding the mono device to multi devices, as seen on the *shape* value ($\{4\}$), allows the model compiler creating an automatic workload distribution of the task and its data. However, an important constraint exists for tasks in such situation: no data overlapping among work-items must exist, neither input (reading) nor output (writing). The host is responsible for scattering and gathering data to/from devices and each device works independently on its data. Usually, this is not a problem because there are many applications that does not require data sharing among

work-items. However, stencil tasks, for instance, cannot easily be used with this distribution approach.

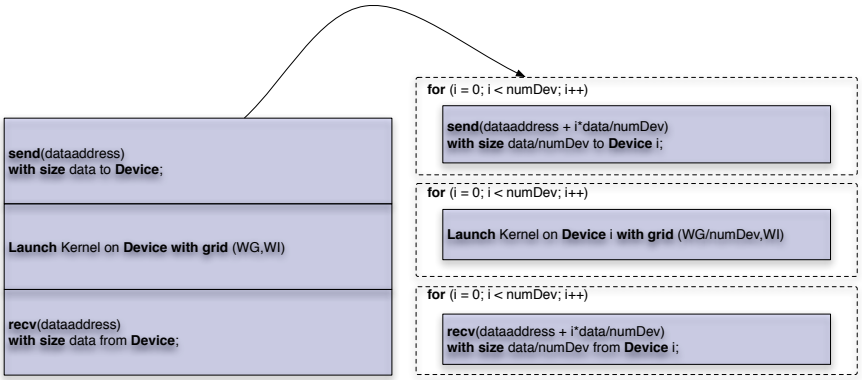


Figure 5.14: Multi-GPU Task Distribution Process

Figure 5.14 illustrates summarily the conversion from mono device to multi device (multi-GPU). Input data are equally distributed to *numDev* devices and then the kernel is launched on each device having a grid definition according with the work-groups distribution [1]. Finally, the host gathers data within a contiguous data structure. All operations are asynchronous among devices.

1. The work-group distribution uses roundup functions to ensure a good workload balance.

5.7 CONCLUSION

In this chapter, we depict the 9 layers that compose the transformation chain. Each layer has a well defined function in the chain. Some of these functions are common aspects found in the most target aimed by the GASPARD2 framework, such as scheduling. Although we have contributed to these common transformation modules, we emphasize, in this chapter, the modules that deal directly with the OpenCL programming model. Thus, memory mapping, hybrid application conception, and the code generation are presented in detail. The model compilation process occurs transparently to model designer. However, this chapter attempts to enlighten essential aspects related to code generation and optimization phases.

6

OPTIMIZATIONS

Chapter Contents

During the development of our hypothesis on code generation, we always had in mind creating high performance solutions. It is not interesting to be able to create applications having low performances compared to "traditional" developing method. Even if our proposal is based on abstract models in a very high-level programming. In this chapter, we present some key points used to optimize the generated code. These points are strongly linked to the running platform, i. e., CPU, GPU under OpenCL programming. Here, we emphasize basically two optimization aspects of the code as described below.

1. **Memory:** data access and data transfers between host and device memories are the main weak points when we work with GPUs. For this reason, in our model compilation process we take into account:
   - possible misplaced data allocations in order to avoid extra data transfers,
   - data reuse by work-groups aiming at exploiting local shared memory.
2. **Profiling:** the most part of the OpenCL APIs give to developers profiling tools. However, there is no interaction between high-level application specification and the results. To fill this gap, we

have proposed a profiling feedback(helped by traceability mechanism) and, additionally, smart advices enlightening bottleneck points in the model.

6.1 MEMORY COPIES

GPUs have dedicated memory which has 5-10x the bandwidth of CPU memory, this is an important advantage. However, beginners in GPU application development are sometimes discouraged by the perceived overhead of transferring data between GPU and CPU memory. The GPU can only be used if the data is moved to it. And similarly, the user interface can only post-process the data if it is moved from the GPU to the CPU. Also, it must be possible to move the data rather fast for high performance computing. In this section, we discuss how an automatic code generator can create applications which do properly data transfers in high throughput conditions, and reduce or eliminate the transfer burden.

In a typical approach, data transfers occur as seen in Figure 6.1. First, copy data from main memory to GPU memory with data throughput of about 10Gbps; second, CPU instructs GPU to start a kernel; third, GPU executes kernel in parallel and accesses GPU memory with throughput about 80Gbps (for typical GPUs); finally, copy the results from GPU memory to CPU main memory. All these events have an expressive completion time compared to the operations on the data. Thus, the analysis of these events become very valuable.

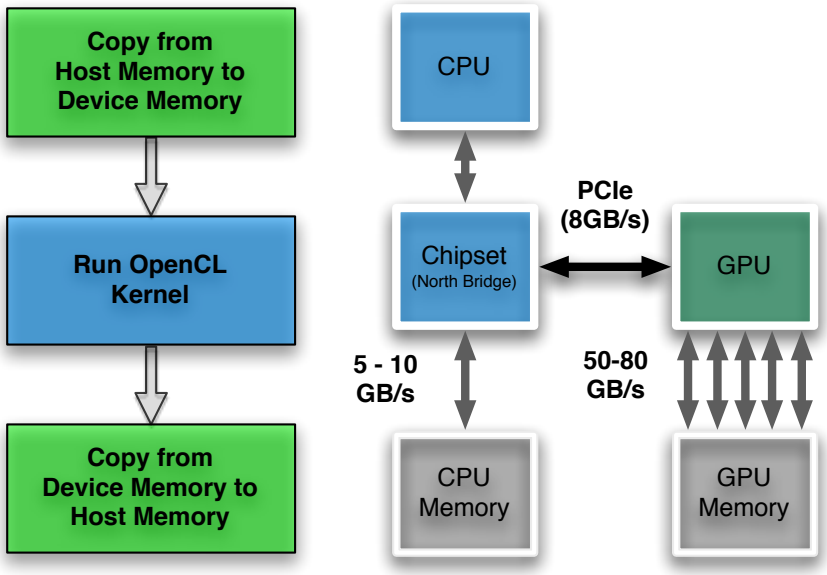


Figure 6.1: Typical Approach with Memory Copy

The OpenCL framework provides a way to package data into a memory object. Using a memory object minimizes memory transfers from the host and device as the kernel processes data. OpenCL memory objects are categorized into two types: buffer objects and image objects. For our approach, we restrict ourselves to buffer objects. They are used to store one-dimensional data such as an *int*, *float*, *vector*, or a user-defined structure. Listing 6.1 depicts the function call `clCreateBuffer()` used to specify buffer objects in the GPU. By using this function one can define a pointer to an allocated buffer in the device memory. This pointer can have specified, for instance, read-only attribute, the allocated space in memory, and where data specified in a pointer of buffer from host will be copied to this allocated buffer. Notice, even though this is a simple procedure, the overhead associated to the data transfer must be taken into account. In some cases, it is interesting to avoid multiple calls of transfer functions such as `clCreateBuffer()`, and, instead, to try to join these calls into only one.

Listing 6.1: `clCreateBuffer()` function call example

```
1 cl_mem input;
2 input = clCreateBuffer(
3       context,    //a valid context
4       CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, // bit-field flag to
5                 // specify the usage of memory
6       sizeof(float) * DATA_SIZE, // size in bytes of the buffer
7       inputsrc,   // pointer of buffer data to be copied from host
8       &err          // returned error code
9 )
```

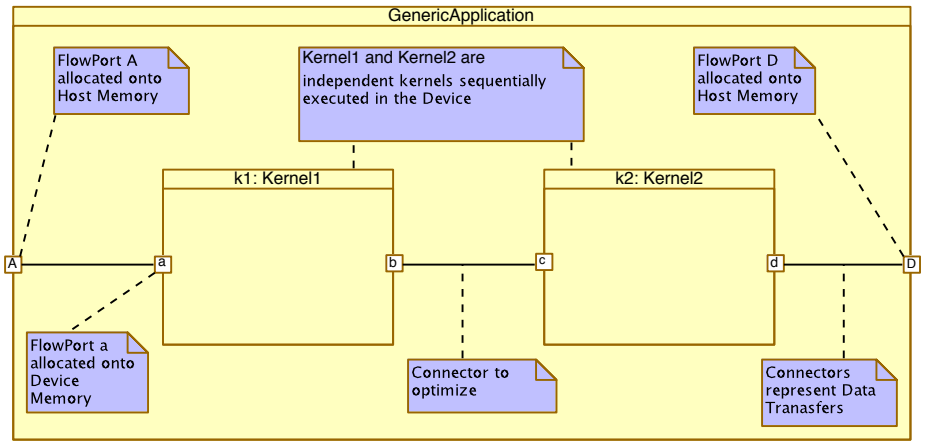


Figure 6.2: Generic Application to illustrate Memory Transfers Suppression

6.1.1 *Avoiding Unnecesary Transfers*

In general, every GPU use by CPU(host) is based on 3 transactions: *send data*, *kernel call*, and *receive data*. However, sometimes multiple kernels can use results previously created by other kernels. Except for the cases where there are tasks performed by the CPU that use a kernel result, there is no need to transfer data back to CPU if subsequent kernels make use of them. Figure 6.2 illustrates this situation. The connector between the *flowports* b and c links two kernel tasks[20] . There is no other connector that links to CPU tasks or tasks performed by different GPU devices. Therefore, the *hybrid* transformation provides a special rule (*cf.* Subsection 5.5.5) that suppress extra data transfers according to model analysis. For the example depicted in Figure 6.2, the transactions are reduced to: *send data*, *kernel1 call*, *kernel2 call*, and *receive data*. This process is applied onto the Downscaler example depicted in Chapter 3.

[20] *These tasks are kernels because they are allocated onto GPU processors (not shown).*

6.2 TILER ANALYSIS

It is common knowledge that the local[21] memory of GPUs is much faster than global memory. Knowing that, we tried to answer the following question: "How can we use *tilers* from ArrayOL to detect when it is worth transfering data from global memory to local memory?"

[21] *Reminder: in OpenCL terminology, local memory is the memory shared among work-items of a same work-group.*

In order to answer this question, we analyze the example depicted in Figure 6.3. In this example, we have a generic kernel *k1* with shape equals `{5,29}`. For our generated code, it will produce 5 work-groups with 29 work-items each one. Each work-item gathers from the global memory a data pattern composed of `{2,2}` elements (port a in the figure) and it scatters `{1}` element (port b). Figure 6.4 presents a diagram of data distribution based on the input and output tilers of *k1*. The figure represents information of the work-items [[0,0], [0,1], [0,13], [0,14]]. The input array (port A) has `{10,29}` elements and the output array (port B) has `{5,29}` elements. For illustration, in Figure 6.5 right side, we present the data collection processed by the work-items of the work-group 1.

6.2.1 *Observing data reuse*

The tiler specification (origin, paving, and fitting) introduced in Chapter B allows us to identify which elements in the input array are processed by each work-item. As an example, for the work-item 0 of the work-group 0, the pattern of 4 elements starting at the position `{0,0}` is used as input (port a). Then, for the work-item 1 of the work-group 0, the pattern starts at the position `{0,2}`, and so on. The next pattern in the work-group always jumps 2 positions in the second dimension. Therefore, analyzing the work-item 14 of the same

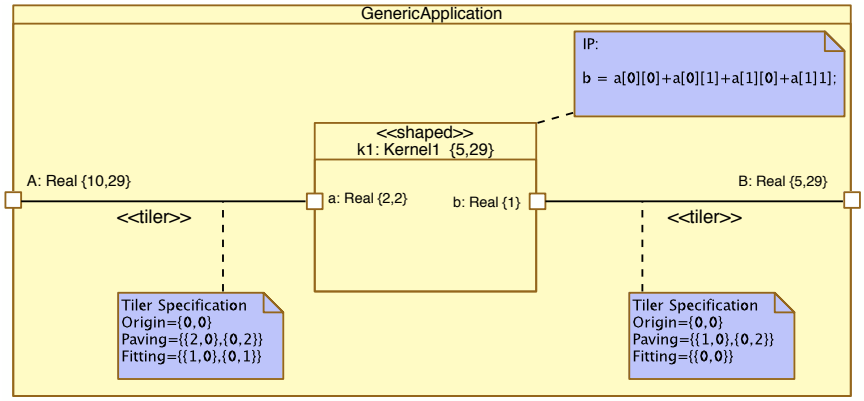


Figure 6.3: Generic Application for Local Memory Optimization

work-group, we see a re-reading of the part of the pattern read by the work-item 0. This fact occurs on all subsequent work-items. For the next analysis, we consider the definitions below.

- $S_{krn}[1..dimkrn]$ is the value of each dimension of the shape of the analyzed kernel, where $dimkrn$ is the total of dimensions of this shape. For our example, $S_{krn}[1..2] = \{5, 29\}$.
- $S_{wi}[1..dimwi]$ is the value of each dimension of the shape of the analyzed work-group of work-items, where $dimwi$ is the total of dimensions of this shape. For our example, $S_{wi}[1..1] = \{29\}$.
- $S_{array}[1..dimarr]$ is the value of each dimension of the shape of the input array, where $dimarr$ is the total of dimensions of this array. $S_{array}[1..2] = \{10, 29\}$ in the example.
- $S_{pattern}[1..dimpat]$ is the value of each dimension of the shape of the input pattern, where $dimpat$ is the total of dimensions of this pattern. $S_{pattern}[1..2] = \{2, 2\}$ in the example.

6.2.2 *Detecting data reuse*

Our hypothesis states that: *if the product between the total of work-items and the total of elements of a pattern is greater or equal to twice the total of elements of the input array, thus, we consider data reuse and, in this case, it is worth transferring data from global memory to local memory.*

This hypothesis can be summarized by the formula:

$$\frac{\prod_{k=1}^{dimkrn} S_{krn}[k] \times \prod_{k=1}^{dimpat} S_{pattern}[k]}{\prod_{k=1}^{dimarr} S_{array}[k]} \geqslant 2 \tag{6.1}$$

Formula (6.1) gives us the degree of data reuse, the bigger this degree, the higher the reuse. This indicates that work-items should copy data from global memory to local memory before starting operations. This copy requires the work-items synchronization in order to

continue working on the data consistently. In the next subsection, we explain our strategy to specify how data will be transferred.

6.2.3 *Deciding which data to transfer*

Through our analysis on *tilers* and data reuse, we came across a crucial question: which data must be copied and who does it? To answer this question we developed the algorithm 1.

Algorithm 1 Optimized Fitting

1: $n \leftarrow cnt\_pnts(Polyhedron(\tilde{P}, F, S_{krn}, S_{arr}, S_{pat}))$ ▷ *From polylib*
2: $\tilde{F} \leftarrow 0$
3: $n_0 \leftarrow cnt\_pnts(Polyhedron(\tilde{P}, \tilde{F}, S_{krn}, S_{arr}, S_{pat}))$
4: **for** $i \leftarrow 1$ **to** $nbcol(F)$ **do**
5: $\quad F_t \leftarrow \begin{pmatrix} \tilde{F} & F_i \end{pmatrix}$
6: $\quad n_t \leftarrow cnt\_pnts(Polyhedron(\tilde{P}, F_t, S_{krn}, S_{arr}, S_{pat}))$
7: $\quad$ **if** $n_t > n_0$ **then**
8: $\quad\quad \tilde{F} \leftarrow F_t$
9: $\quad\quad n_0 \leftarrow n_t$
10: $\quad$ **end if**
11: **end for**

The idea behind this algorithm is attempting to find the minimal `fitting` $\tilde{F}$ that ensures the reading of the total of elements of a work-group. In each iteration, the algorithm increases the columns of the `fitting` matrix. Afterwards, it stores the column which leads a change in the number of points belonging to the polyhedron as seen in line 1,3, and 6. To count the number of points we use the polylib library [128].

As an example for the data in Figure 6.3, Formula (6.1) gives:

$$\frac{(5 \times 29) \times (2 \times 2)}{10 \times 29} = 2$$

This result allows us considering the local(shared) memory use. Applying the algorithm 1 in the input tilers definition:

1) $\tilde{P} = \begin{pmatrix} 0 & 0 \\ 0 & 2 \end{pmatrix}$, work-group paving;
2) $n = 116$, all readings by a work-group;
3) $n_0 = 29$, `fitting matrix` is null: $\tilde{F} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$;
4) $F_t = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$, only column 1 $\implies$ $n_t = 58$;
5) $n_t > n_0$ is true $\implies$ $\tilde{F} = F_t$;
6) $F_t = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, columns 1 and 2 $\implies$ $n_t = 58, n_0 = 58$;
7) $n_t > n_0$ is false;

At the end, $\tilde{F} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$. This `fitting` replaces the original `fitting` for each work-item within a work-group. Each work-item copies data from global memory to local(shared) memory and executes a synchronization operation.

That is not the general solution to the problem. Indeed, work-items could read diagonal elements as well, but the `fitting` that gives the diagonal indexes is not considered in algorithm 1. This requires in-depth matrix analysis. However, this algorithm works in most cases and shows the potential of ArrayOL in the study of cache handling. This is particularly important to architectures such as GPU due to their memory hierarchy.

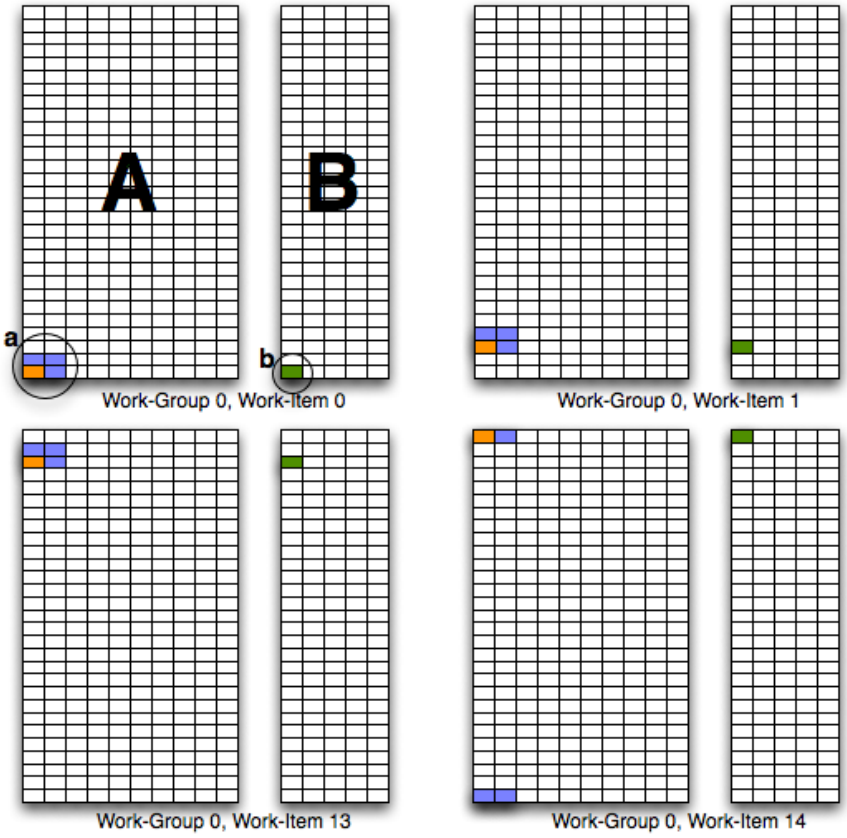


Figure 6.4: Input and Output Arrays and Patterns for Work-Group 0

6.3 PROFILING ANALYSIS

The main goal in this analysis is to provide a high-level profiling environment in a model design context. Performance execution feedbacks are directly provided in the input models. In a such way, the

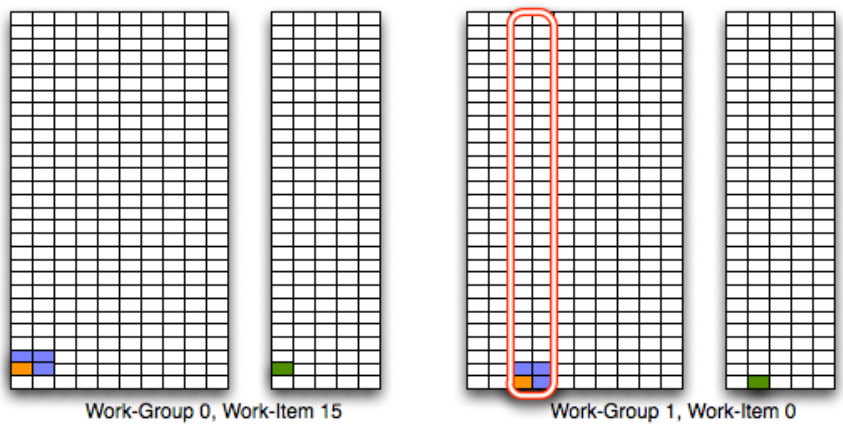


Figure 6.5: Different Polygons depending on Work-Groups

model designers can easily understand the designed system behavior and identify what should be fine tuned. This approach takes advantage of real data from a profiling tool, instead of using performance statically computed from performance constraints designed in the input models.

The approach is sketched in Figure 6.6. More details of the whole optimization approach can be found in the technical report [130]. The profiling lifecycle presented follows a classic structure generally manually performed:

1. the software is generated from the high-level models (step 1) and the trace models are produced according to trace mechanism found in [6, 5, 7];
2. the software is executed, producing profiling logs (steps 2 to 4);
3. the produced logs are analyzed and returned in the input models (steps 5 to 7).

Currently, in works available through the literature, only the first part is an automatic process (step 1). The second part (steps 2 to 4) which produces the logs depends strongly on the used tools. The third part whose logs are analyzed and connected to the input elements that should be modified remains a manual and complex process. We have focused on this sub-process (steps 5 to 7 in Figure 6.6) aiming to automate it.

In order to automate this analysis and the performances feedback, we have proposed two key modules: an expert system (reported as *Domain Specific Profiling Analysis Transformation Library* in Figure 6.6) and the Model-to-Model Transformation (M2M) traceability (*Trace Model*). The expert system is used during the analysis step, whereas the traceability is used for the performance feedback.

At first, we present how the traceability is managed in the compilation chain and the required modifications on the model compilation

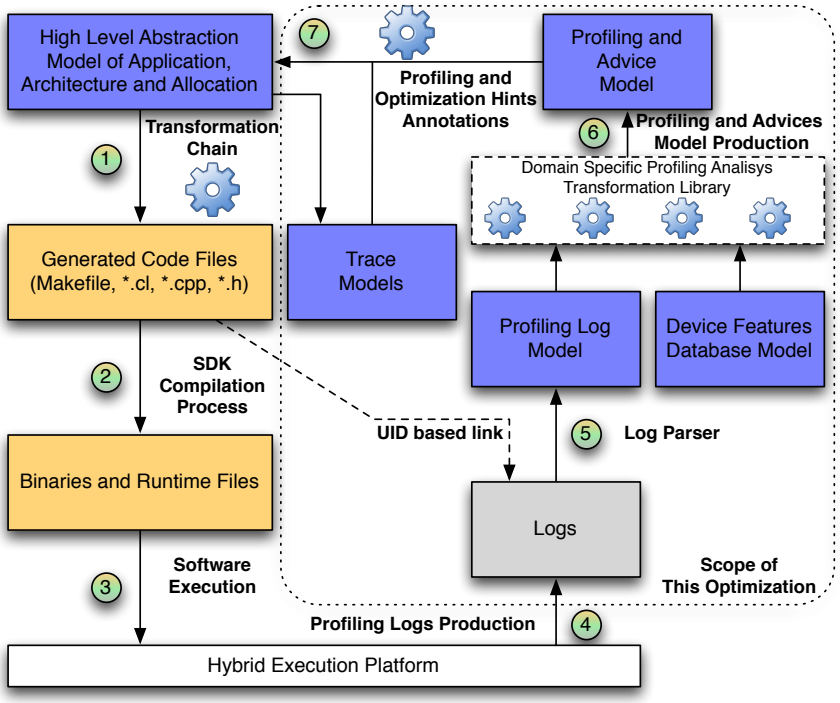


Figure 6.6: Performance and Profiling Integration Overview

chain. Afterwards, we provide the expert system to create the profiling advices and, finally, we show how these advices are reported into input models.

6.3.1 *Managing The Whole Chain Traceability and Avoiding Model-to-Text Traceability*

In order to keep the links between the input models and the software execution, trace models are produced all along the compilation chain, except for the model-to-text transformation. The translation from model to text implies keeping information on text blocks and words [118]. The granularity for this kind of trace made its management and maintainability difficult. In our case, the code has to be studied only in terms of the abstract concepts from the models, and not in terms of blocks and words. Thus, we avoid the model-to-text traceability.

To bypass the model-to-text trace, the code generation deals with unique identifiers (UIDs) associated to each elements in the last model. The profiling logs produced by the software execution refers to the UID of the analyzed element. Thus, the *Profiling Logs* can rebound to the model world.

Concretely, in order to generate the UIDs, we use the EMF feature called *Universal Unique IDentifier* (UUID) and, consequently, we mod-

ify the compilation chain. A new transformation adding the UID was inserted as last step of the model-to-model transformation chain, just before the code generation.

6.3.2 *From Execution to Smart Advices*

Once the software code is generated, the software is executed. During the execution, profiling logs are produced by third party tools.

6.3.2.1 *Profiling Logs Parsing*

According to the used Software Development Kit (SDK) and profiling tools, these profiling logs are generated with a dedicated format. This format is parsed using a shell-script that builds a profiling model that conforms to the metamodel presented in Figure 6.7.

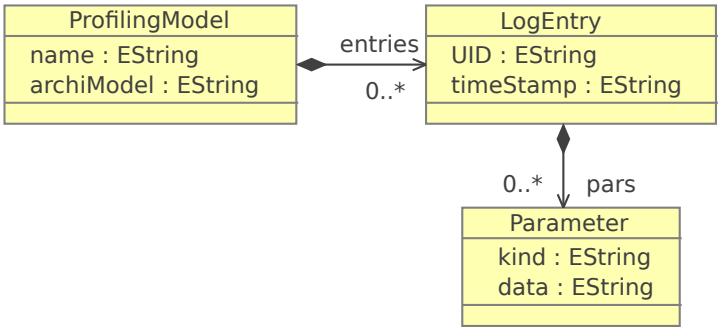


Figure 6.7: Profiling Metamodel

The metamodel root: *ProfilingModel* gathers the different entries that can be found in the profiling logs. Each profiling entry from the logs is represented by a *LogEntry* gathering the hardware model (*e.g.* Tesla T10 or G80) with the *archiModel* attribute. Each *LogEntry* contains several *Parameter* elements owning a *kind* and a *data* representing: the information type (e. g., occupancy, time or memory consumption), and its value. In order to keep the link between the profiling information and the transformation chain, each *LogEntry* keeps the UID from the logs with the *UID* attribute. In addition, a *timeStamp* attribute is added to the *LogEntry* in order to keep the logs sequence. This metamodel is generic enough to produce models that can gather information that can be found in the profiling logs.

6.3.2.2 *The Expert System*

In-depth knowledge of the target platform is essential to identify which elements should be modified. Model designers do not always have such a knowledge. Thus, more than profiling results, we propose to provide smart advices to model designers. To reach this aim, we integrate an expert system that uses input data from two sources:

profiling logs and *device features database*. The first one gives us factual data about execution. The second one gives us behavioral features of the target platform. By combining both sources, it is possible to deduce what to do to attain some optimization level. For instance, assuming the device supports 32MB in shared memory allocation per group of work-items and the application allocates at runtime 48MB. The expert system is able to indicate that it is necessary to decrease the memory allocation after analyzing profiling log results and device constraints. In this case, the expert system provides a hint where the problem occurs. In order to analyze the many properties of the results, an extensible library (*cf.* Figure 6.6) is proposed.

The device features database gathers a set of devices from a specific vendor. It is represented as a model that conforms to the metamodel in Figure 6.8 in order to be properly handled by either transformation languages or other tools based on the Eclipse Modeling Framework (EMF). In the context of the UML/MARTE-to-OpenCL transformation chain, the target platform is the GPU.

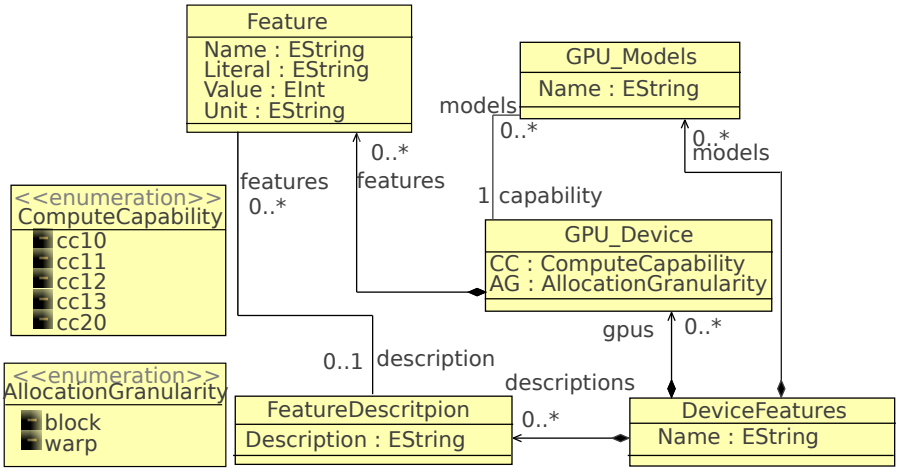


Figure 6.8: GPU Device Features Metamodel

The model root is represented by *DeviceFeratures*. It gathers the many vendors' GPU models (*e.g.*, Tesla T10) represented by the *GPU_Models* concept. These GPU models are associated to a group of GPU devices (represented by the *GPU_Device* concept) having the same allocation granularity (*AG* attribute) and compute capability (*CC* attribute). Their values are specified according to two enumerations: *AllocationGranularity* and *ComputeCapability*, respectively. In the first enumeration, each literal represents the compute capability version of the GPU (*e.g.*, the literal *cc10* represents the 1.0 compute capability version). In the second enumeration two choices are possible: the data allocation is performed either by *block*(work-groups) or by *warp*[22] .

The different devices presented in the model contain various features and their descriptions (represented by *Feature* and *FeatureDescrip-*

[22] *Reminder: a* ***warp*** *is a GPU related concept that indicates the number of work-items in a work-group ready to hardware scheduling. It is also known as wavefront.*

tion). So, for instance, the GPU Tesla T10 which has compute capability 1.3 has the feature *Work-Items per Warp* equals 32. Figure 6.11 in Subsection 6.3.4 shows a model based on this metamodel.

6.3.3 *Backtracking Advices in the Input Models*

The advice as well as the computed value obtained during the software execution must be reported in the input model. Thus, the model designer can directly access the advice reported on the element requiring the modification. For the information feedback, the reduced trace produced during the transformation chain execution is used.

The UID contained in the computed advice refers to elements in the last model before the code generation. This UID comes from the *Profiling Logs* and it is retained even after transformations. Once the element referred by the UID is found, the reduced trace is backward navigated in order to recover the input elements producing the profiling information. Two cases can occur: the retrieved elements are reduced to one or to several elements. Indeed, a simple element in the last model can be produced from either one or many elements in the input models.

Reporting the advice on all retrieved elements makes the result complex. To solve this issue, the expert system can be configured to specify some element types of the input metamodel for each library. Therefore, only the input elements of the specified types are kept. Finally, the computed advice is connected to these elements in the input models.

In the UML/MARTE-to-OpenCL transformation chain, we decide to use the *Comment* concept from UML. Indeed, this concept element has the ability to gather information in a string format. Moreover, *Comment* can be linked to any kind of UML element what places it as a perfect candidate for carrying the advice computed by the expert system. As the advice representation is quite dependent from the input metamodel and the input metamodel capabilities, it could be provided using an other concept.

6.3.4 *Example and Benchmarks*

In order to illustrate the practical application of this optimization, we describe a case study. This case study is a complete example which presents an application design, code generation and the profiling feedback.

In this example the goal is to offer information and compute an advice about how to improve the processors' occupancy. This will help the application designers to identify input elements parameters which they can modify aiming better results.

6.3.4.1 *Vector Product Application*

The example presented in this section is a vector product. For presentation reasons, we chose this simple operation that does not require further knowledge in more complex applications such as signal processing or numerical analysis to be understood. Nevertheless, it gathers all the relevant concepts to illustrate our approach. Moreover, we have also tested our approach on large scale examples like the classic downscaler algorithm [112]. The vector product is an algebraic operation that takes two equal-length(N) sequences of numbers and return another sequence obtained by multiplying corresponding entries. A sequential code sample in C language for this operation is showed in the listing 6.2.

Listing 6.2: Code Sample for Vector Product

```
1  for (uint i=0; i<N; i++)
2   c[i] = a[i] * b[i];
```

We have created the UML-MARTE model (see Figure 6.9) for this application using the Papyrus [24] modeling tool. Elementary tasks *TE_genarray1*, *TE_genarray2*, and *TE_printarray* are responsible for generating and printing vectors. As these tasks compose the application interface (data input and output), and the CPU performs them.

The application's vectors are arrays of 16,000,000 elements. We have chosen this large number to take advantage of the massively parallel processors provided by GPUs. The composed component *ForDevices* instantiated in the program consists of a repetitive task *ep:TE_elemprod*. In our application, this kind of task is composed by operations on single elements. Repetitive tasks are potentially parallel and are allocated onto GPU. The repetition shape of the task in this case is *{16,1000000}*, *i.e.* the task operation runs 16 millions times on one element of each vector whose size equals 16 millions. This shape has two dimensions: 16 and 1,000,000. The total of work-items is calculated by multiplying these two dimensions. The first one becomes the number of work-items and the second one the number of groups. The definition of this shape is a decision of the designers and usually they take into account the IP interface associated to the elementary task and its external *tilers*. Moreover, considering that, in compute capability 1.x devices, memory transfers and instruction dispatch occur at the Half-Warp(16 work-items) granularity, it is reasonable to define groups composed by 16 work-items at a first try.

Figure 6.10 presents the allocation process for the repetitive task, *i.e.* it defines which devices will manage a task. For instance, the *ep: TE_elemprod* task (also visible in Figure 6.9) will be run by the GPU (*gpu: GPU* instance). Similarly, the memory mapping process is also defined for the communication ports. According to Figure 6.10, the communication ports will be managed by the GPU memory (*gpumem: GPUMEM*). Thanks to the task allocation process, the model compiler

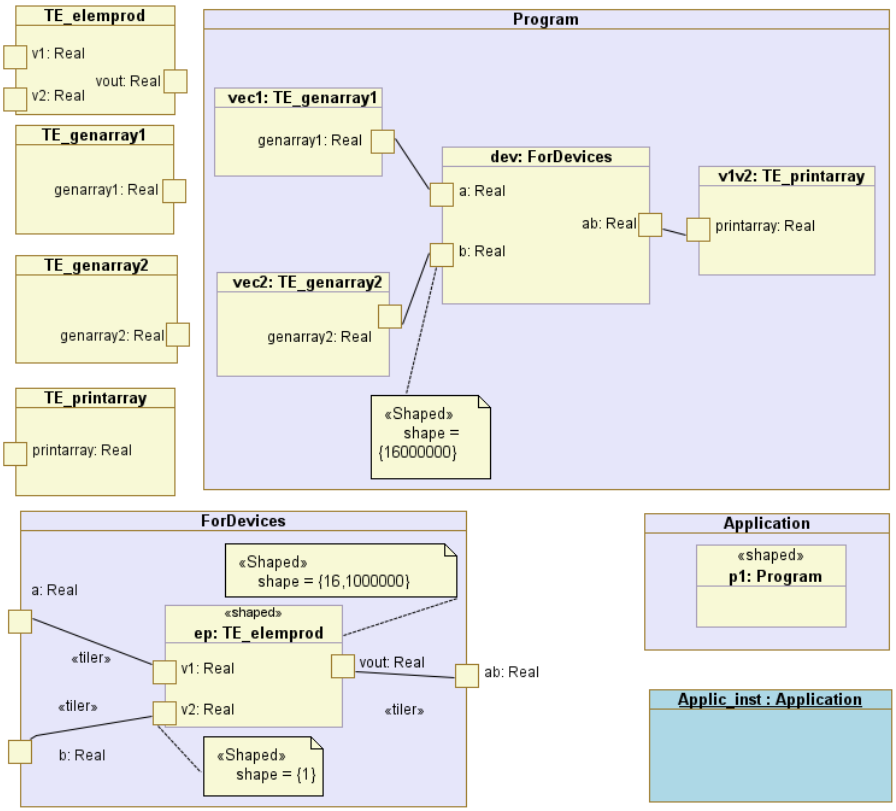


Figure 6.9: Vector Product Application Model

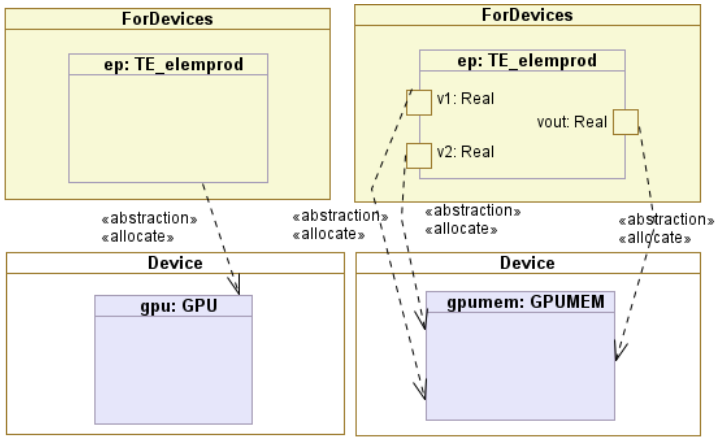


Figure 6.10: Task and Memory Allocations onto GPU

identifies *OpenCL kernels*. The memory allocation step is also important because it creates *host* and *device* variables and organizes the data transfers.

Once the application is designed with all necessary well configured elements, we generate all the source code files necessary to the target compiler. In addition, trace models are generated for each model-to-model transformation thanks to the traceability mechanism.

Listing 6.3 shows the code only for the *kernel*. This is a generated code composed of two functions: the *IP* function, represented in lines 1-4, and the *kernel* function (lines 6 to 60). The UID value (see Section 6.3.1) is identified from the last generated model in the UML/MARTE-to-OpenCL transformation chain and is concatenated to the *kernel* name. From line 6 we identify, concatenated to *kernel* name, the UID *_uCQs6obGEeCiXMyak_whYg*. This UID is the link between the code and the model elements whenever they have to be referenced.

Listing 6.3: Generated Kernel for Vector Product

```
1  void elemprod(const float* a, const float* b, float* c)
2  {
3    c[0]=a[0]*b[0];
4  }
5
6  __kernel void ep_KRN__uCQs6obGEeCiXMyak_whYg(
7  uint iNumElements,
8  const __global float* v2_ep_KRNPAR,
9  __global float* vout_ep_KRNPAR,
10 const __global float* v1_ep_KRNPAR)
11 {
12   float v1_ep[1]; float v2_ep[1]; float vout_ep[1];
13   //get index into globaldata array
14   int iGID = get_global_id(0) +
15   get_global_size(0)*get_global_id(1) +
16   get_global_size(0)*get_global_size(1)*get_global_id(2);
17   if (iGID < iNumElements) // bound check
18   {
19     { //input tiler
20       uint tlIter[2];
21       uint tl[1];
22       uint ref[1];
23       uint index[1];
24       tlIter[0]=iGID%16;
25       tlIter[1]=abs(iGID/16);
26       ref[0] = 0 + 1*tlIter[0] + 1*tlIter[1]*tlIter[0];
27       for(tl[0]=0; tl[0] < 1; tl[0]++) {
28         index[0]= (ref[0]+ 0*tl[0])%16000000;
29         v2_ep[tl[0] * 1] = v2_ep_KRNPAR[index[0] * 1];
30       }
31     }
32     { //input tiler
33       uint tlIter[2];
34       uint tl[1];
35       uint ref[1];
36       uint index[1];
```

```
37        tlIter[0]=iGID%16;
38        tlIter[1]=abs(iGID/16);
39        ref[0] = 0 + 1*tlIter[0] + 1*tlIter[1]*tlIter[0];
40        for(tl[0]=0; tl[0] < 1; tl[0]++) {
41          index[0]= (ref[0]+ 0*tl[0])%16000000;
42          v1_ep[tl[0] * 1] = v1_ep_KRNPAR[index[0] * 1];
43        }
44      }
45      elemprod(v1_ep, v2_ep, vout_ep); //IP call
46      { //output tiler uint
47        tlIter[2];
48        uint tl[1];
49        uint ref[1];
50        uint index[1];
51        tlIter[0]=iGID%16;
52        tlIter[1]=abs(iGID/16);
53        ref[0] = 0+ 1*tlIter[0] + 1*tlIter[1]*tlIter[0];
54        for(tl[0]=0; tl[0] < 1; tl[0]++) {
55          index[0]= (ref[0]+ 1*tl[0])%16000000;
56          vout_ep_KRNPAR[index[0] * 1]=vout_ep[tl[0] * 1];
57        }
58      }
59    }
60  }
```

The remaining code consists of private variable declarations (line 12), a limit control to avoid overlapping data bounds (lines 14-17), two input *tilers* to gather the elements from global memory (lines 18-43), the *IP* call (line 45); and an output *tiler* writing the result into global memory (lines 47 to 56).

6.3.4.2 *Profiling Feedback*

We used the following configuration as platform for our tests:

- CPU AMD Opteron 8-core @2.4GHz and 64GB RAM;
- GPU NVidia S1070 4 devices Tesla T10 (4GB RAM each) - Compute Capability 1.3;
- Linux, GCC 4.1.2, OpenCL 1.0.

Among all the measures coming from the profiler, the kernel occupancy factor has an important impact on performance. Usually the aim at executing a kernel is to keep the multiprocessors and, consequently, the device as busy as possible. The work-items instructions are executed sequentially in OpenCL, and, as a result, executing other warps. When one warp is paused or stalled is the only way to hide latencies and keep the hardware busy. Some metrics related to the number of active warps on a multiprocessor is therefore important in determining how effectively the hardware is kept busy. This metric is called occupancy. The occupancy is the ratio of the number of active warps per multiprocessor (WPM) to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware ability to process warps that are actively in use. Hence,

higher occupancy does not always equate to higher performance, there is a point above where additional occupancy does not improve performance. However, low occupancy always interferes with the ability to hide memory latency, resulting in performance degradation.

The important features to compute the occupancy and that vary on the different compute capability are:

- the number of registers available;
- the maximum number of simultaneous work-items resident on each multiprocessor;
- and the register allocation granularity.

The number of work-items resident on a multiprocessor relies on index space as known as *N-Dimensional Range (NDRange)*. The MARTE to OpenCL chain obtains the information from the shape of the task which will become a *kernel*. Hence, changes in the dimensions of shape affect the occupancy. From the point of view of the proposed approach, *occupancy* is a specialized module that can be included to the expert system. For other analysis other specialized module can be added to attain specific goals. For this example, we analyze the *occupancy* of the multiprocessors. Occupancy is a function of constant parameters (features) from device and some measures directly obtained from the profiler.

The process of calculating occupancy is implemented in a QVT transformation. This transformation takes two input models (according to Figure 6.6): the *Device Features Database* and the *Profiling Logs*. In this example the first one conforms to a metamodel based on NVidia GPUs (*cf.* Figure 6.8). Although this metamodel was designed according to vendor's features, it can be modified or extended to comply with other vendor device models. For instance, from the model presented in Figure 6.11 we see that the GPU Tesla T10 has compute capability equals 1.3 and is its warps contain 32 threads (or work-items in OpenCL terminology).

6.3.4.3 *Benchmark*

The profiling environment creates a log file in CSV format having some dynamic measured data (as seen in Figure 6.12). The file header contains data about the target platform. We deal with a Tesla T10 GPU in this case. The rest of the file consists in description of fields and log entries. The description of fields indicates in which order they will appear in a log entry. For instance, in Figure 6.12, a log entry begins with the *timestamp* field. The second field that can be retrieved in an entry is *gpustarttimestamp*, then *method* and the 13-th field that can be found in the log is the *occupancy* field. For our example, log entries about memory copies are not important (log entry with the *method* field sets to *memcpyHtoDasync*). The following entries in the log correspond to *kernel* calls. A shell-script parser takes this text file as input and converts it to XMI format that conforms to the profiling

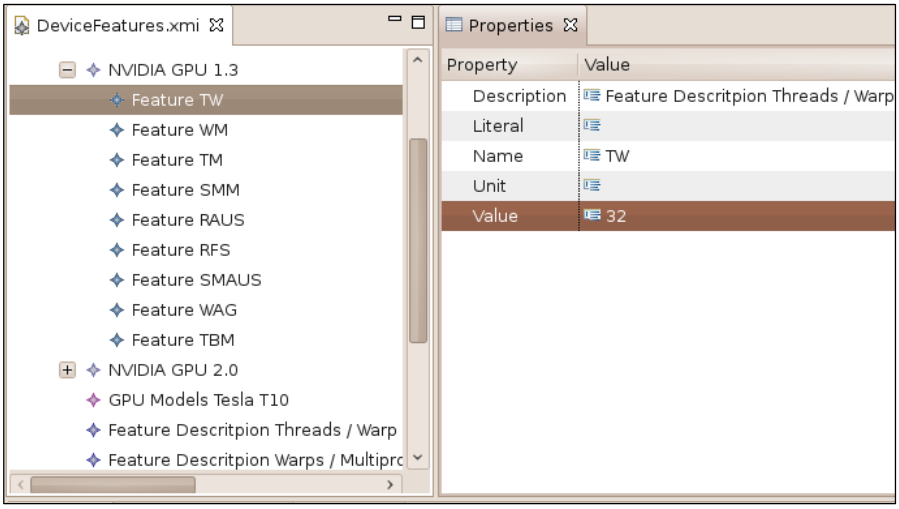


Figure 6.11: GPU Device Features Database Model

metamodel depicted in Figure 6.7. The model (Figure 6.13) created from the CSV log file gathers exactly th same information. Except the *timestamp* field and the UID contained in the *method* field that becomes attributes of the *LogEntry*, all the other fields (*e.g. gputime*, *cputime* or *ndrangesizex*) are transformed into *Parameter* with the value of the log entry. Figure 6.12 (highlighted elements) and Figure 6.13 present the occupancy parameter with value 0.250 for the kernel call with timestamp equals 1283955.000.

```
# OPENCL_PROFILE_LOG_VERSION 2.0
# OPENCL_DEVICE 0 Tesla T10 Processor
# OPENCL_PROFILE_CSV 1
# TIMESTAMPFACTOR fffff6f3dd57cd20
timestamp,gpustarttimestamp,method,gputime,cputime,ndrangesizeX,ndranges
izeY,workgroupsizeX,workgroupsizeY,workgroupsizeZ,stapmemperworkgroup,re
gperworkitem,occupancy,streamid,local_load,local_store,gld_request,gst_r
equest,memtransfersize,memtransferdir,memtransferhostmemtype
698239.000,1220f847c305ce40,memcpyHtoDasync,40347.039,41886.000,,,,,,,,,
1,,,,,64000000,1,0
1241546.000,1220f847e367f960,memcpyHtoDasync,40539.457,41379.00
0,,,,,,,,,1,,,,,64000000,1,0
1283955.000,1220f847e5e6a2c0,ep_KRN__uCQs6obGEeCiXMyak_whYg,1238.752,147
0.000,65535,1,16,1,1,32,11,0.250,1,0,0,4370,2185
1285577.000,1220f847e5fee540,ep_KRN__uCQs6obGEeCiXMyak_whYg,1237.120,142
8.000,65535,1,16,1,1,32,11,0.250,1,0,0,4368,2184
1287138.000,1220f847e616bd40,ep_KRN__uCQs6obGEeCiXMyak_whYg,1237.600,143
7.000,65535,1,16,1,1,32,11,0.250,1,0,0,4370,2185
...
```

Figure 6.12: Sample profiling results in CSV format

Although the methodology that integrates profiling and high-level models does not impose a rigid workflow, it relies on two major activities: first we run the code exactly as it is generated from the original input model, then the application designer analyzes the runtime behavior based on profiling feedback annotated on the input model;

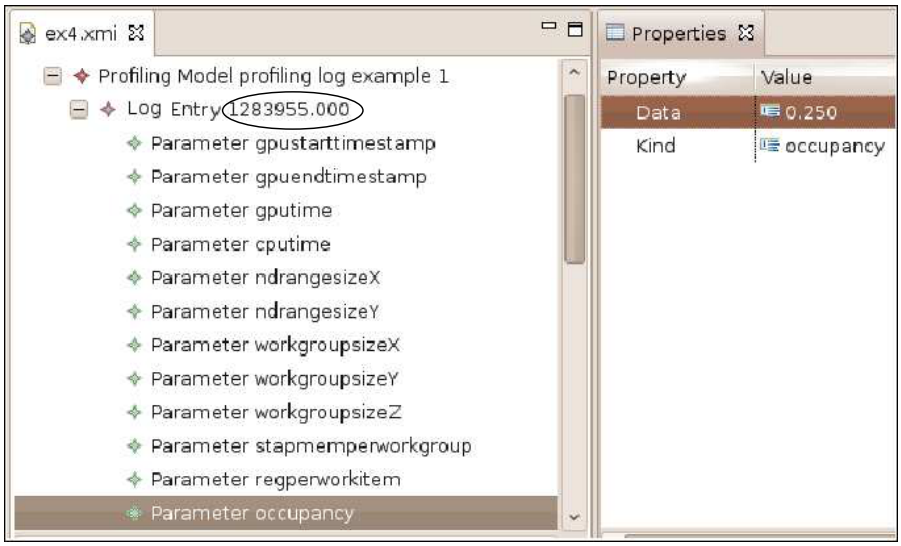


Figure 6.13: Profiling Results Model

second, the designer, taking into account the provided information and hints, modifies the model aiming to obtain better results. Once the model is modified, the code is again generated and then executed in order to verify the result of theses changes.

For this example the first running gives the results seen in Table 6.1. The application launches 1 million groups of 16 work-items onto the device. However, a hardware limit imposes a maximum of 65535 groups by kernel. Thus, for the whole execution, 15 kernels are launched with the maximum of 65535 groups and 1 kernel is launched with 16975 groups. Summarizing, the last line in table shows that this configuration gives 16 kernels calls and only 25% in the multiprocessor occupancy. Moreover, this configuration take 2.55% of the GPU time. The other part of the time comprehends data transfers and idle states. The launch grid (NDRange) has a two-dimensional size (*i.e.*, how many groups and how they are organized). Each group has a three-dimensional size (represented by *[16 1 1]* on third column). However, only the first dimension is used in this example. Our goal is to increase the occupancy and decrease the relative GPU time.

Table 6.1: Profiling results for the non-optimized code

Calls #	NDRange	WGSize	Occup.	GPUTime
15(each one)	[65535 1]	[16 1 1]	25%	-
1	[16975 1]	[16 1 1]	25%	-
16	[1000000 1]	[16 1 1]	25%	2.55%

By using our approach results are combined with GPU features and this returns a smart advice as a comment in the input UML-MARTE

model (Figure 6.14). Besides the performance parameters available directly on the comment, a *hint* points out a possible change in the model to improve the generated code. Additionally, the advice provides an image reference of a chart (as seen in Figure 6.15) for all predicted occupancies according to these results. In this case it is suggested to change the task shape from $\{16, 1000000\}$ to $\{128, 125000\}$. A simple analysis seeks the first block size giving 100% on occupancy as seen in Figure 6.15. For instance, the expert system automatically highlights the first (block size=128) and second (block size=256) maximum values in the chart.

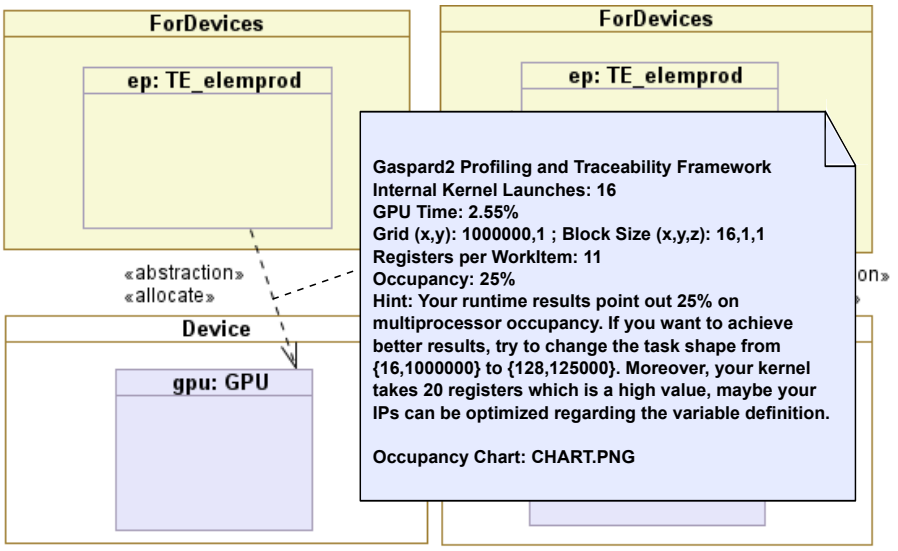


Figure 6.14: Annotated Model

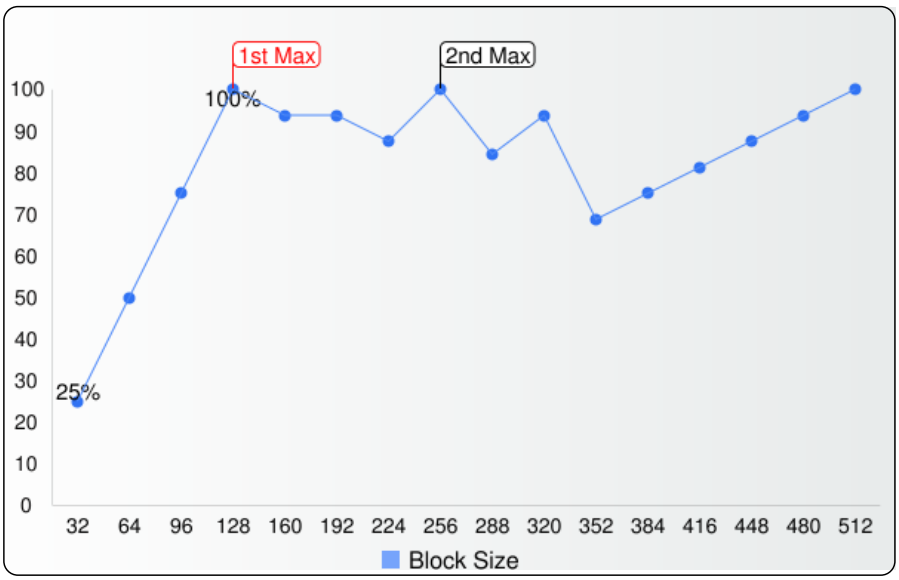


Figure 6.15: Occupancy by Varying Block Size

Table 6.2: Profiling results for the new code

Calls #	NDRange	WGSize	Occup.	GPUTime
1	[65535 1]	[128 1 1]	100%	-
1	[59465 1]	[128 1 1]	100%	-
2	[125000 1]	[128 1 1]	100%	1.07%

Table 6.2 presents the profiling results for the code generated from the modified input model. For this case, we have two kernel calls, 100% on occupancy and a reduction to 1.07% on the GPU time. As expected, the modified model achieves better performance than the original one. Figure 6.16, obtained from a visual profiler provided within NVidia tools, shows us that, without modifications, the whole execution of the kernel is about 146% slower.

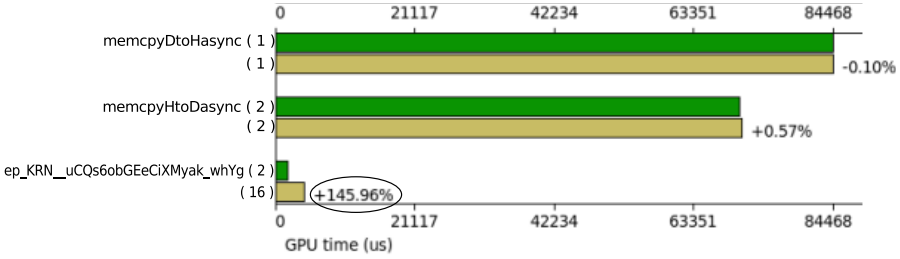


Figure 6.16: Comparison Summary Plot from Visual Profiler

For each operation, the first bar is related to the time measured from the optimized models, whereas the second bar is related to the time measured from the initial models. The transfer times presented in Figure 6.16 correspond to the following operations:

- *memcpyDtoHasync* is called once in both executions. This transfers the result vector from device to host.
- *memcpyHtoDasync* is called twice in both executions. This transfers the two input vectors from host to device.

As we have changed only the task shape, transfer times do not have any expressive alteration.

6.4 CONCLUSION

Our general objective is to provide a methodology that generates efficient code. This depends mainly on the designed model of application. However, we show in this chapter some optimization concerns that the transformation chain can analyze and also propose means to improve the generated code. In summary, the methodology provides two resources. First, one of the most crucial concerns in GPU: memory use. Indee, the suppression of extra data transfers and the *tiler* analysis

on data reuse help to fill the requirements for memory issues. Second, the integration between profiling results and high-level models allows the model designer to understand the runtime behavior. Moreover, advices lead designers to tune their models in order to achieve better performances. Finally, we consider these resources helpful for the co-design environment.

Part III

SIMULATION OF ELECTRICAL SYSTEMS

7

ELECTROMAGNETIC PHENOMENON AND CODE_CARMEL

Chapter Contents

We have worked with the Laboratory of Electrical and Power Electronics of Lille (L2EP) on high-level abstraction models of numerical methods of simple problems such as an electric field induced by a changing magnetic field or more complex problems such as the simulation of electrical machines (e. g., automotive alternators). This chapter and the next one present a case analysis of our methodology on problems of electromagnetism. The scientific computation associated to the solution of these problems requires numerical methods in the discrete domain. So, in order to assure a better understanding of the decision taken in the case study, at first, we present part of the theory of the mathematical modeling of these problems in the continuous domain. Then, we introduce the FEM as numerical method to solve these problems.

Having this basic theory, we present the Code Avancé de Recherche pour les Machines Électriques (Code_CARMEL). This code was originally and is still developed in Fortran 90. However, our contributions allowed to extend it to accept other languages such as C and, consequently, OpenCL.

7.1 LAWS OF ELECTROMAGNETISM

Electromagnetism (EM) is concerned with the study of the relationship between electricity and magnetism. Classical electromagnetism (or classical electrodynamics) is a branch of theoretical physics that studies consequences of the electromagnetic forces between electric charges and currents. It provides an excellent description of electromagnetic phenomena whenever the relevant length scales and field strengths are large enough that quantum mechanical effects are negligible. The theory of electromagnetism was developed over the course of the 19th century, most prominently by James Clerk Maxwell. Other great scientists played an important role in this field, for a detailed historical account, consult [52].

James Clerk Maxwell (13 June 1831 in Edinburgh 5 November 1879 in Cambridge) was a Scottish physicist and mathematician.

7.1.1 *Continuous-time Maxwell's Equations*

Maxwell unified the laws of electromagnetism into what is known today as Maxwell's Laws or Maxwell's Equations. These equations first stated in [102]. They are comprised of a set of Partial Differential Equations (PDE). Maxwell's equations represent one of the most elegant and concise ways to state the fundamentals of electricity and magnetism. From them, one can develop most of the working relationships in the field. Maxwell's equations were created by combining the laws of Karl Friedrich Gauss, Ampere and Michael Faraday with the discovery by the professor Hans Christian Oersted in 1820. He also added a hypothesis of his own. The theory of electromagnetic field founded by Faraday was mathematically completed by Maxwell. One of the most new ideas put forward by Maxwell was the idea of symmetry in the mutual dependence of electric and magnetic fields. Namely, since a time-varying magnetic field ($\partial \mathbf{B}/\partial t$) creates an electric field, it should be expected that a time-varying electric field ($\partial \mathbf{E}/\partial t$) creates a magnetic field.

Maxwell's Laws may be summarized as follow:

- The first one states that the electric field has a source called charge.
- The second law states that the magnetic field does not have an equivalent source; there are no single magnetic poles called "monopoles."
- Finally, laws three and four state that a changing magnetic field produces an electric field, and a changing electric field or current produces a magnetic field. Law four is Maxwell's contribution.

7.1.1.1 *Space-Time Domain*

The quantities in Maxwell's equations are defined on a 4-dimensional space-time domain. Here, we only discuss their form in an inertial

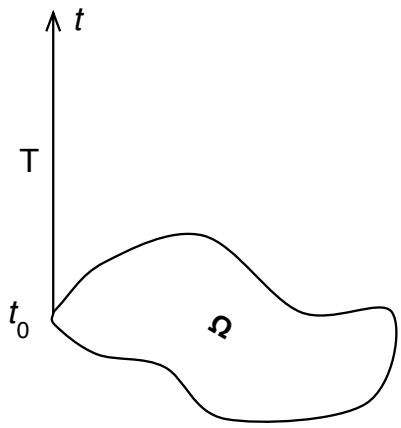


Figure 7.1: Continuous problem domain ΩxT. *Source: Euler's Thesis [53]*

mode and separate the space-time into the 3-dimensional spatial domain Ω and the 1-dimensional temporal domain T. The problem domain is regarded as the Cartesian product of these spaces ΩxT as illustrated in Figure 7.1. The spatial variable will be denoted by $\vec{r} \in \Omega$ and the temporal variable by $t \in T$. This setting is suitable for most engineering applications and allows for the separate treatment of space and time in the discretization process. The so-called local forms of the Maxwell's equations are the following:

$$\nabla \times \mathbf{H} = \mathbf{J} + \frac{\partial \mathbf{D}}{\partial t} \tag{7.1}$$

$$\nabla \times \mathbf{E} = \frac{-\partial \mathbf{B}}{\partial t} \tag{7.2}$$

$$\nabla \cdot \mathbf{B} = 0 \tag{7.3}$$

$$\nabla \cdot \mathbf{D} = \rho \tag{7.4}$$

where,

H is vector of the magnetic field strength (A/m);

J is the current density vector (T);

D is the electric displacement vector (V/m^{-1});

E is the electric field strength (C/m^2);

B is the magnetic induction vector (C/m^3);

ρ is the charge density.

Equations (7.1), (7.2), (7.3), and (7.4) are, respectively, *the generalized Ampère's law*, *Faraday's law of induction*, *the Gauss's law for magnetism*, and *the Gauss's law* in their differential form. Although the electromagnetism phenomena were established by other scientists before Maxwell, there was some incompatibility on the formulation and

Maxwell, by introducing an additional term to the *the generalized Ampère's law*, could synthesize the electromagnetism in 4 equations. As an interpretation of Equations (7.1) and (7.2), the variation either of the magnetic or electric fields in relation to time can generate each other. This phenomenon is known as electromagnetic coupling.

7.1.1.2 *Electromagnetism Division for Physical Applications*

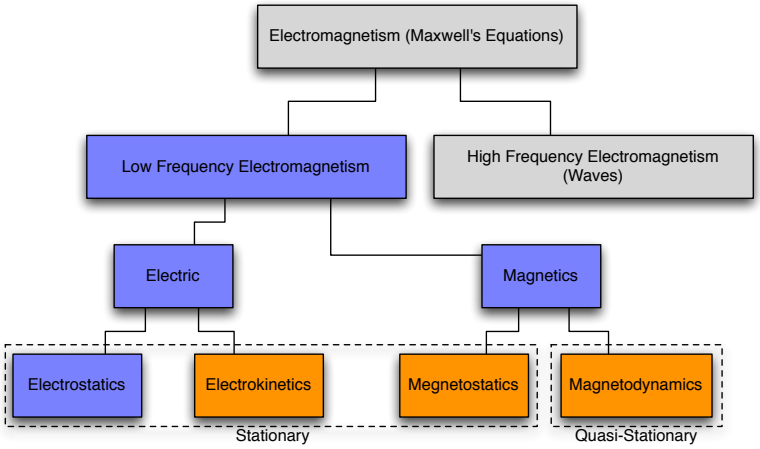


Figure 7.2: Electromagnetism division for physical applications. In each block of this diagram, the 4 Maxwell's equations are adapted according to the corresponding physical situation.

The electromagnetism is divided into two parts: *low frequency electromagnetism* and *high frequency electromagnetism*. The last one is divided into two parts: the *stationary*, where there is no time variation, and the *quasi-stationary* or magnetodynamics.

For low frequencies, the 4 Maxwell equations can be simplified omitting certain terms regarding the dynamics of the process. For quasi-stationary fields (e. g. alternating current), for instance, where the current $\mathbf{J} \neq 0$ and the magnetic field is changing $\frac{\partial \mathbf{B}}{\partial t} \neq 0$, the rate of changing of the electric field can be neglected $\frac{\partial \mathbf{D}}{\partial t} \approx 0$. In this case, Equation (7.1) can be simplified to the following form:

$$\nabla \times \mathbf{H} = \mathbf{J} \tag{7.5}$$

7.1.1.3 *Constitutive Relations: Simpler Case*

In order to apply Maxwell's macroscopic equations, it is necessary to specify the relations between displacement vector $\mathbf{D}$ and electric field $\mathbf{E}$, and the magnetic field $\mathbf{H}$ and induction vector $\mathbf{B}$. These equations specify the response of bound charge and current to the applied fields and are called constitutive relations. These relations, also known as behavior laws, express the material properties. If μ represents the

permeability of free space, ε the permittivity of free space, and σ the electrical conductivity, the behavior laws are as follow:

$$\mathbf{B} = \mu\mathbf{H} \tag{7.6}$$

$$\mathbf{D} = \varepsilon\mathbf{E} \tag{7.7}$$

$$\mathbf{J} = \sigma\mathbf{E} \tag{7.8}$$

Equation (7.8) is also known as Ohm's law. Here, we consider the simpler case where there is absence of magnetic or dielectric materials. Substituting these back into Maxwell's macroscopic equations (7.1)-(7.4) leads directly to Maxwell's microscopic equations, except that the currents and charges are replaced with free currents and free charges. This is expected since there are no bound charges nor currents.

7.1.1.4 *Boundary Conditions*

Like all sets of differential equations, Maxwell's equations cannot be uniquely solved without a suitable set of boundary conditions and initial conditions. These conditions must be specified in order to obtain a unique solution for the fields. Maxwell's equations, given sources $\mathbf{J}$ and ρ, together with an appropriate boundary conditions, constitute a problem that can be uniquely solved for unknown field quantities.

Two types of boundary conditions are most currently used:

- Dirichlet boundary conditions: it specifies the values a solution needs to take on the boundary of the domain.
- Neumann boundary condition: it specifies the values that the derivative of a solution is to take on the boundary of the domain.

The choice between the one of the two approaches depends on the problem.

7.2 DISCRETIZATION: FEM

The analytical solution of Maxwell's equations allows to obtain a mathematical exact solution of the problem. However, it is not suitable to more complex problems and it is limited to simpler configurations. As an example, Dodd et al. [47] worked in direct solutions to the differential equations using the separation of variables method. In their work, we can find the following configuration:

1. A solution for a coil above a semi-infinite conducting slab with a plane surface, covered with a uniform layer of another conductor. This solution includes the special cases of a coil above a single infinite plane conductor or above a sheet of finite thickness, as well as the case of one metal clad on another.
2. The other solution is for a coil surrounding an infinitely long circular conducting rod with a uniformly thick coating of another

conductor. This includes the special cases of a coil around a conducting tube or rod, as well as one metal clad on a rod of another metal.

As expected for an analytical solution, their conclusion claims the agreement between the calculated and experimental values is excellent. Although more recent works [141, 142] present solutions for a coil with magnetic core on a plane surface or an inclined air core coil on a plane surface, more complex configurations make the analytical solution either impossible or they requires very high computational resources. On the other hand, we can adopt numerical methods which allow to provide solutions with negligible errors.

7.2.1 *Method*

The Finite Element Method (FEM) is a computational method to solve boundary value problems over an unstructured mesh. FEM is particularly well suited for modeling domains of arbitrary shape, and efficiently modeling small features in large computational domains.

In the FEM, the solution domain is divided or discretized into small regions called "finite elements" [11, 82]. For 2D applications these elements can be triangles, for instance. The corners that define the triangle (see Figure 7.3) are the nodes or degrees of freedom. The set of these elements is called mesh.

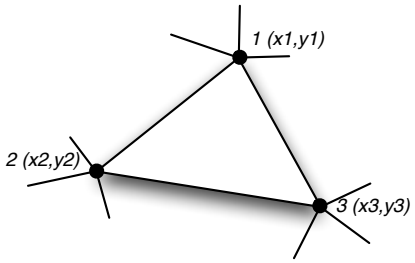


Figure 7.3: An Element in a Triangular Mesh

7.2.2 *Assembly and Solvers*

Usually, the computation of the numerical solution involves two steps: the assembly of matrix $\mathbf{A}$ and vector $\mathbf{b}$, and the solution of the linear system for $\mathbf{x}$ ($\mathbf{Ax} = \mathbf{b}$). The assembly process depends on the problem. Nevertheless, usually the process takes each element K from the mesh and builds a so-called elementary matrix of size $(N_v + N_e) \times (N_v + N_e)$, where N_v is the number of vertices and N_e

the number of edges. This elementary matrix $\mathbf{A}^K$ consists of 4 blocks: $\mathbf{A}^K = \begin{pmatrix} \mathbf{A}^K_{11} & \mathbf{A}^K_{12} \\ \mathbf{A}^K_{21} & \mathbf{A}^K_{22} \end{pmatrix}$.

We build the vector $\mathbf{b}$ similarly the matrix $\mathbf{A}$. It is composed of elementary vectors $\mathbf{b}^K$ of size $N_\nu + N_e$, $\mathbf{b}^K = \begin{pmatrix} \mathbf{b}^K_1 \\ \mathbf{b}^K_2 \end{pmatrix}$.

To solve the resulting linear system, many direct and iterative methods are available [69, 10] as seen in lists below:

- Direct solvers: LU, MUMPS, SuperLU, Cholesky, QR, so on.
- Iterative solvers: SOR, SSOR, GMRES, Chebychev, Richardson, conjugate gradients (Cg), CGSquared, BiCgStab, two variants of TFQMR, conjugate residuals, Lsqr, so on.

These lists are huge. Here we are interested on the iterative solver: Conjugate Gradient (CG). It is our case study in following chapter.

7.3 THE CODE_CARMEL

7.3.1 *Introduction to* CODE_CARMEL

Code Avancé de Recherche pour les Machines Électriques (CODE_CARMEL) is a code initially developed by the Modeling Team from L2EP laboratory [1]. Today, this code is also maintained by the modeling on electromagnetism team from Électricité de France (EDF) R&D. It is a code based on Finite Element Method (FEM) that allows to solve electrocinetics, magnetostatics, or even magnetodynamics problems. The code is currently written in Fortran 90, and it is in constant evolution.

1. l2ep.univ-lille1.fr

7.3.2 *Formulations*

CODE_CARMEL supports two types of formulation [16]:

- $(\mathbf{A}, \Phi)$;
- $(\mathbf{T}, \Omega)$;

7.3.2.1 *The* $(\mathbf{A}, \Phi)$ *formulation*

In the $(\mathbf{A}, \Phi)$ formulation, the magnetic vector potential $\mathbf{A}$ and the electric scalar potential Φ are unknown. The unknown degrees of freedom (*dofs*) of $\mathbf{A}$ are defined on the edges of the whole model. The unknown *dofs* of Φ are only defined at the vertices of the conducting domains. There is no uniqueness in the choice of the $(\mathbf{A}, \Phi)$ unknowns. In order to ensure uniqueness, a gauge condition regarding $\mathbf{A}$ and Φ must be imposed.

Gauge condition in Code_CARMEL for $(\mathbf{A}, \Phi)$ formulation [2]: The relationship between $(\mathbf{A}, \Phi)$ potentials and the electric and magnetic field is given by:

2. Remark: In the current version, it is not possible to explicitly enforce a gauge condition.

$$\begin{cases} \mathbf{E} &= -i\omega\mathbf{A} - \nabla\Phi \\ \mathbf{B} &= -\nabla \times \mathbf{A} \end{cases}$$

7.3.2.2 *The* $(\mathbf{T}, \Omega)$ *formulation*

We consider the conducting domain $\mathcal{D}_C$, of the mesh model, as the union of all conductors in the mesh model. We define the insulator domain $\mathcal{D}_I$ by: $\mathcal{D}_I = \mathfrak{R}^3 \setminus \mathcal{D}_C$. $\mathbf{N}$ is global divergenceless physical current and it is generated by the global vector field $\mathbf{K}|\nabla \times \mathbf{K} = \mathbf{N} \in \mathfrak{R}^3$.

In the $(\mathbf{T}, \Omega)$ formulation, the unknowns are the current vector potential $\mathbf{T}$ and the magnetic scalar potential Ω. The unknown degrees of freedom (*dofs*) of $\mathbf{T}$ are defined on the edges of the conducting domains only. The unknown *dofs* of Ω are defined at the vertices of the whole domain.

There is no uniqueness in the choice of the $(\mathbf{T}, \Omega)$ unknowns. In order to ensure uniqueness, a gauge condition must be imposed. In the formulation, this gauge condition consists in prescribing the continuation of the magnetic scalar potential Ω within the conducting domains.

Gauge condition in Code_CARMEL for $(\mathbf{T}, \Omega)$ formulation [1]:

1. Remark: In the current version, it is not possible to explicitly enforce a gauge condition.

The relation between the $(\mathbf{T}, \Omega)$ potentials and the electric and magnetic fields is less straightforward than in the $(\mathbf{A}, \Phi)$ formulation due to the topology of the conducting domain $\mathcal{D}_C$ plays a role. The total magnetic field $\mathbf{H}$ is written as the sum of a magnetic source field $\mathbf{H_s}$ and a magnetic reaction field $\mathbf{H_r}$: $\mathbf{H} = \mathbf{H_s} + \mathbf{H_r}$

The magnetic reaction field writes:

$$\mathbf{H_r} = \begin{cases} -\nabla\Omega + \sum_{j=1}^{p} \kappa_j \mathbf{K_j} & \text{in } \mathfrak{R}^3 \setminus \mathcal{D}_C \\ \mathbf{T} - \nabla\Omega + \sum_{j=1}^{p} \kappa_j \mathbf{K_j} & \text{in } \mathcal{D}_C \end{cases}$$

The electric field is only known in the conducting domain $\mathcal{D}_C$ by: $\mathbf{E} = \sigma^{-1}\nabla \times H_r$.

7.3.3 *Running* Code_CARMEL *in Parallel*

Making a parallel version of Code_CARMEL was already part of the goal of the Julien Taillard and Emmanuel Cagniot's thesis [137, 21].

In [21], Cagniot developed two new versions of Code_CARMEL. High Performance Fortran (HPF) [89] and Halos [14] were the tools used to create a "light" and a "heavy" version. The light one is based on HPF directives directly modifying the original code. This method implies on about 1% of changes, thus called "light". The second one, the heavy one, implies algorithm changes. This approach intends to assure the load balance among the available processors and can be implemented on distributed systems.

Taillard, on his turn, proposed OpenMP [121] as parallel approach to Code_CARMEL. His approach is based on high-level models based on UML (as already presented in Chapter 1).

The constant evolution of CODE_CARMEL and the lack of a well defined methodology to integrate the parallel modules are the factors that avoid the full parallel version portability.

The popularity and availability of GPUs and the integration method used in the case study seen in next chapter provide means to assure the constant evolution of the parallel code.

7.3.4 *Global Structure*

CODE_CARMEL is composed of several phases of computation. Each phase has an executable module for which the user can interact. Figure 7.4 presents the *use case diagram* of CODE_CARMEL. The user can independently call each executable module providing input data as runtime parameters. Globally, the user has interactivity with the modules below. We summarize them according to their function and typical order of use based on the type of problem:

- `GENDOF` is responsible to build the finite elements model. It reads a mesh file defined as input data. This file has `.med` format type (produced by SALOME [25].
- `GENPHYS` is responsible for the definition of the physical parameters and the source. It reads the file generated on the earlier module (`gendof`) and creates a `.phys` file.
- `GENPARAM` is responsible for the definition of the computation parameters. Based on the previously defined files, it generates a `.param` file necessary to `FCARMEL`.
- `FCARMEL` comprehends the linear system assembly and its resolution. Its output is the solution for the system (`.xmat` file).
- `POSTPROCESS` reads every file previously produced and writes out:
  - global quantities (flows through the inductors, loss of energy, magnetic energy);
  - fields maps of $\mathbf{E}, \mathbf{J}, \mathbf{H}$, and $\mathbf{B}$, ohmic loss density, K and $\nabla \times K$ by source, total $N + \nabla \times K$ saved as 3D fields in `.med` format (SALOME) for time problems only. These fields are complex or real for harmonic or time problems, respectively;
  - fields maps of $\mathbf{E}, \mathbf{J}, \mathbf{H}$, and $\mathbf{B}$, on cutting planes;
  - values of fields $\mathbf{E}, \mathbf{J}, \mathbf{H}$, and $\mathbf{B}$, on cutting planes.

The modules, described above, are the same in all situations, except the assembly/solver where we have `FCARMEL`(fcarmel.exe) for harmonic problems and we replace it for `TCARMEL`(tcarmel.exe) for time-dependent problems.

In Figure 7.5, we have the activity diagram for CODE_CARMEL. This diagram represents the workflows of stepwise activities and actions of the earlier presented use cases. Here, it gives us only the right execution order.

The sequence diagram is given in Figure 7.6. This diagram is a kind of interaction diagram that shows how processes operate with one

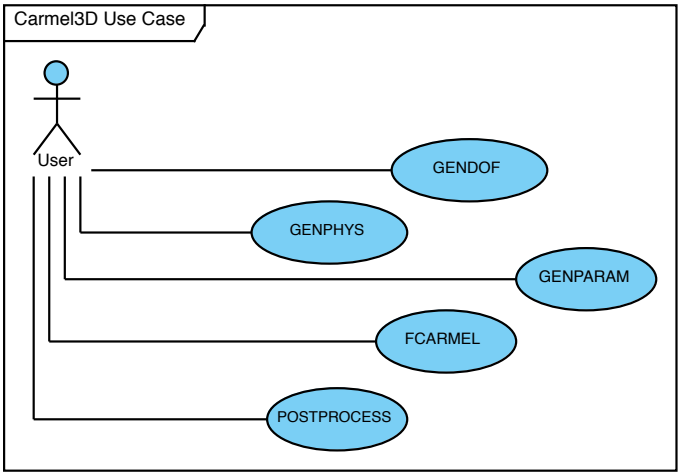


Figure 7.4: UML Use Case Model for the code_CARMEL3D

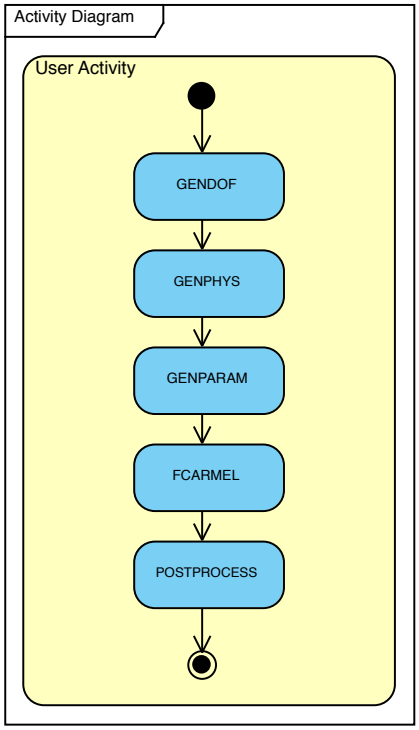


Figure 7.5: UML Activity Diagram Model for code_CARMEL3D

another and in what order. It depicts the actor and modules involved in the execution scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. This diagram provides also event and timing information. Indeed, from the point of view of the user, the `FCARMEL` module takes more time than the others. More precisely, this module is sub-divided into two modules: *matrix assembly* and *solver*. Both involve many repetitive tasks to achieve their objective. These modules are potentially parallel tasks and deserve attention when deciding to reduce overall computation times.

The user chooses the solver to be used. Currently, Code_CARMEL offers the solvers BiConjugate Gradient Conjugate Residual (BiCGCR) for harmonic problems and Conjugate Gradient (CG) for time problems. Although it is necessary to do some changes on the code in order to add other solver modules, Code_CARMEL allows to use external solvers such as MUltifrontal Massively Parallel sparse direct Solver (MUMPS) [28], a direct solver, GMRES [28], and the preconditioned CG [1] [131], iterative solvers.

1. The Conjugate Gradient in its non-preconditioned version is subject of the next chapter.

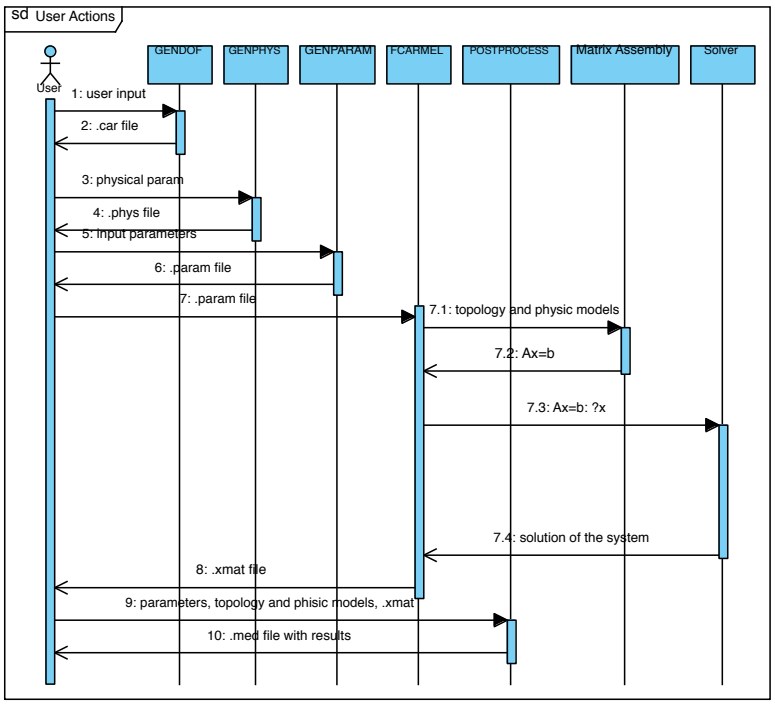


Figure 7.6: UML Sequence Diagram Model for code_CARMEL3D

7.4 CONCLUSION

In this chapter, we presented the basis of electromagnetism phenomenon and the Code Avancé de Recherche pour les Machines

Électriques (CODE_CARMEL). The concepts presented here are important to the understanding the electrical machine simulation. The simulation approach implemented on CODE_CARMEL is supported by the Maxwell's equations. Moreover, two modules of CODE_CARMEL take a lot of time depending on the problem. For these situations, parallel approaches are well suitable. Next chapter deals with the application the methodology presented in this thesis to the solver module in CODE_CARMEL.

8

CONJUGATE GRADIENT SOLVER

Chapter Contents

In this chapter, we present the direct application of the methodology exposed in this work to the parallelism of a solver in the Code_CARMEL context. The objectives can be summarized into two main aspects: First, the high level specification of the solver's algorithm using MARTE. Second, the expected results by applying our solver module into the Code_CARMEL. Further, with this case study, we can demonstrate one of the main features of our whole approach: the integration between software and hardware. Indeed, when specifying GPU devices, we can modify the device multiplicity. This impacts directly the whole code and provides an automatically generated multi-GPU code. Consequently, better performances are expected.

At first, we explain the solver's algorithm chosen for our case study as emphasized in the previous chapter. This concerns the Conjugate Gradient (CG) in its version without preconditioner. Afterwards, we present its model, code generation and benchmarks. Moreover, we describe two issues related to the CG method: parallel reduction in dot product and sparse matrix format concerns regarding matrix vector product.

At the end, we illustrate our approach applied to a real-world example, an automotive alternator from Valeo[23] .

[23] www.valeo.fr

8.1 INTRODUCTION TO CONJUGATE GRADIENT

Basically, we are interested in providing a function to solve the linear system:

$$Ax = b \tag{8.1}$$

where $A \in \mathbb{R}^{n_x n}$ is large, sparse, symmetric, and nonsingular; x and b are vectors $\in \mathbb{R}^n$. If A is positive definite, then the linear system can be solved using the CG method of Hestenes and Stiefel [79]. Unlike matrix factorization, the CG method depicted here for solving (8.1) regards A as an operator and only requires matrix-vector products, building up x as a combination of vectors derived from a Krylov sequence [88].

Algorithm 2 has exactly the same structure of the final code that we are looking for. Basically, it is divided into two parts: the initialization phase where we prepare auxiliary variables for the second part, the main loop. Usually the algorithm converges in n (number of unknowns) or less iterations. However, we stop the loop when the error achieves an acceptable tolerance. Each iteration is composed of:

- 1 *matrix x vector* operation for Ap_k in lines 8 and 10;
- 3 scalar product (or dot product) operations in lines 8 and 11;
- 3 *AXPY* ($y = \alpha x + y$) operations in lines 9,10, and 12;
- some scalar operations as seen in lines 8,11, and 13.

Algorithm 2 CG without Preconditionner

1: $x_0 \leftarrow 0$
2: $r_0 \leftarrow b$
3: $norm_{r0} \leftarrow norm2(r_0)$
4: $p_0 \leftarrow r_0$
5: $error \leftarrow 1$
6: $k \leftarrow 0$
7: **while** $error >$ ERROR_MAX **do** ▷ We stop if error is sufficiently small
8: $\quad \alpha \leftarrow \frac{(r_k^T r_k)}{(p_k^T A p_k)}$
9: $\quad x_{k+1} \leftarrow x_k + \alpha_k p_k$
10: $\quad r_{k+1} \leftarrow r_k - \alpha_k A p_k$
11: $\quad \beta \leftarrow \frac{(r_{k+1}^T r_{k+1})}{(r_k^T r_k)}$
12: $\quad p_{k+1} \leftarrow r_{k+1} - \beta_k p_k$
13: $\quad error \leftarrow \frac{norm(r)}{norm_{r0}}$
14: $\quad k \leftarrow k + 1$
15: **end while**

8.1.1 *Sparse Matrix*

Those operations are highly parallel, even if the product *matrix x vector* uses *sparse* [64][24] matrix, typical from this kind of problem. Our sparse matrices have a special format that stores the non-zero elements by rows according to Compressed Sparse Row (CSR) format. The CSR

[24] *Sparse matrix is a matrix populated primarily with zeros*

or CRS format has its column array normally stored ahead of the row index array. CSR is represented by three arrays (val, ja, ia), where val is an array of the (left-to-right, then top-to-bottom) non-zero values of the matrix; ja is the column indices corresponding to the values; and ia is the list of value indexes where each row starts. The name is based on the fact that row index information is compressed. This format is fairly efficient for arithmetic operations, row slicing, and matrix-vector products.

As an example:

A matrix is given as $A(nxm) = \begin{pmatrix} 1 & 2 & 0 & 0 \\ 0 & 3 & 9 & 0 \\ 0 & 1 & 4 & 0 \end{pmatrix}$, where $n = 3$ and $m = 4$, the CSR format for A is:

```
A = [ 1 2 3 9 1 4]; ja = [1 2 2 3 2 3]; ia = [1 3 5 7].
```

The array A has nnz (total of non-zero) elements as well as array ja. The array ia has $n + 1$ elements and its last element is always $nnz + 1$. Specially for our symmetric matrices, we can see the original matrix as only the diagonal and either the upper or lower part. This reduces in almost 50% the storing space for this matrix. Nevertheless, due to memory access constraints in GPU, we do not use the symmetric store format.

The whole algorithm, taking into account the above presented format, is designed as UML/MARTE model. Next section presents this case study.

8.2 CASE STUDY

This section deals with all steps involved in specification, generation and result analysis. Particularly, we emphasize here the multi-GPU specification and its impacts on the last example presented in this case study.

8.2.1 *High-Level Specification*

As seen in Chapter 3, UMLs/MARTEs within modeling tools is used to specify our application and architecture at high-level abstractions. Figure 8.2 shows the global view of our application. The CG is built as a solver module for Code_CARMEL as seen in Figure 8.1. The idea behind the model that we will provide is that it will replace the original solver written in Fortran90 by a GPU solver in OpenCL. This process is a simple step in the compilation process of Code_CARMEL. Back to Figure 8.2, we have three components: *init*, *norm_ro*, and *cg*. The enlightened component *norm_ro* is a repeated task and is allocated onto the GPU processor. The *init* is an instance of the

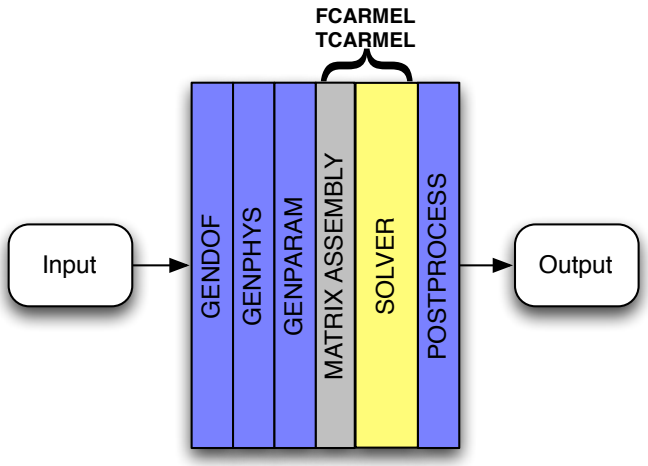


Figure 8.1: Usual Modules of Code_CARMEL

class *InitVars* that initializes the vector x_0. The *cg* is the component that implements the main loop seen in Algorithm 2. Most of its parameters come from the external environment to CG solver and from the Code_CARMEL environment. Moreover, this repeated task presents some new elements provided by MARTE but not yet discussed:

- **«defaultlink»** stereotype allows to point out the data value used in the first iteration of a repeated task.
- **«interrepetition»** stereotype allows to redefine input variables by output variables from previous iteration. This creates a dependency between iterations and avoids the parallelism. However, in the CG application model, the repeated task is usually executed in sequential model by the CPU.
- **«NFP_Constraint»** stereotype allows to add a non-functional property to our component. Here it applies a constraint to ensure the loop continuation: *(error > ERROR_MAX)*. *NFP_Constraint* extends the UML mechanism for applying a condition or restriction to modeled elements. Specifically, NFP constraints support textual expressions to specify assertions regarding performance, scheduling, and other embedded systems' features, and their relationship to other features by means of variables, mathematical, logical, and time expressions.

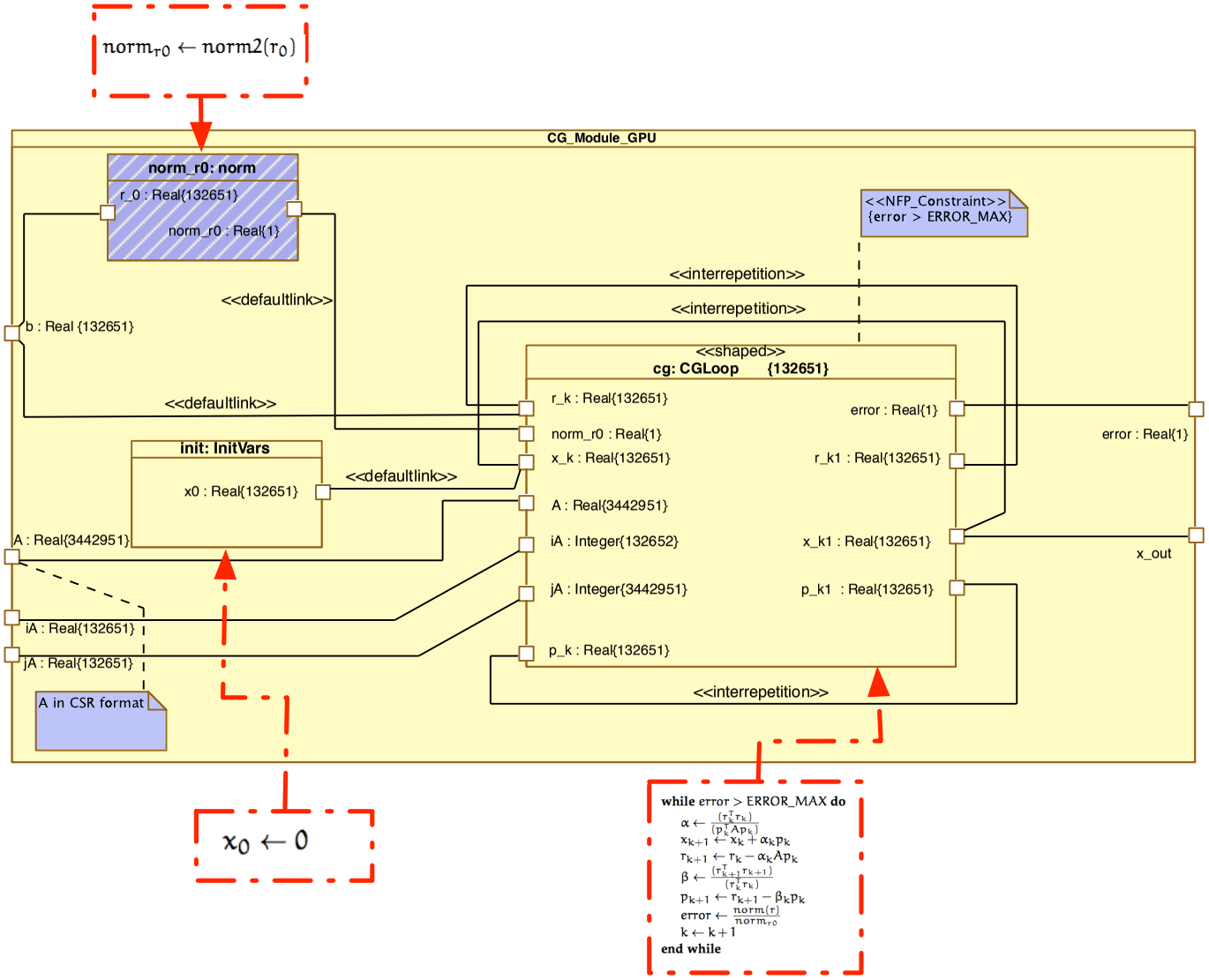


Figure 8.2: Conjugate Gradient Global View

The internal structure of the *cg* composed task is seen in Figure 8.3. This figure does not present all the designed details. Indeed, it shows the essential information for this model. For example, the *Ap* task is an instance of *dgemvCSR* component and it has a **«shaped»** stereotype. Actually, this task takes 4 arrays and makes the matrix-vector product for CSR matrices, where each instance of the repetition space takes one row of matrix A and multiplies by a vector p_k.

Even if all procedures for an application modeling as seen in Chapter 3 are necessary, we emphasize here only the task allocation. For this modeling approach, the model designer should allocate all *shaped* tasks onto the device GPU and the scalar operations (as well as the parallel reductions) are allocated onto the host CPU (as partially seen in Figure 8.3). This is the fundamental principle necessary to identify all produced kernels.

In both Figures of CG model, we emphasize the corresponding algorithm operation. This helps in model understanding. Moreover, dot products are considered special tasks (highlighted ones with diagonal-lines pattern) and the following paragraph describes them in more detail.

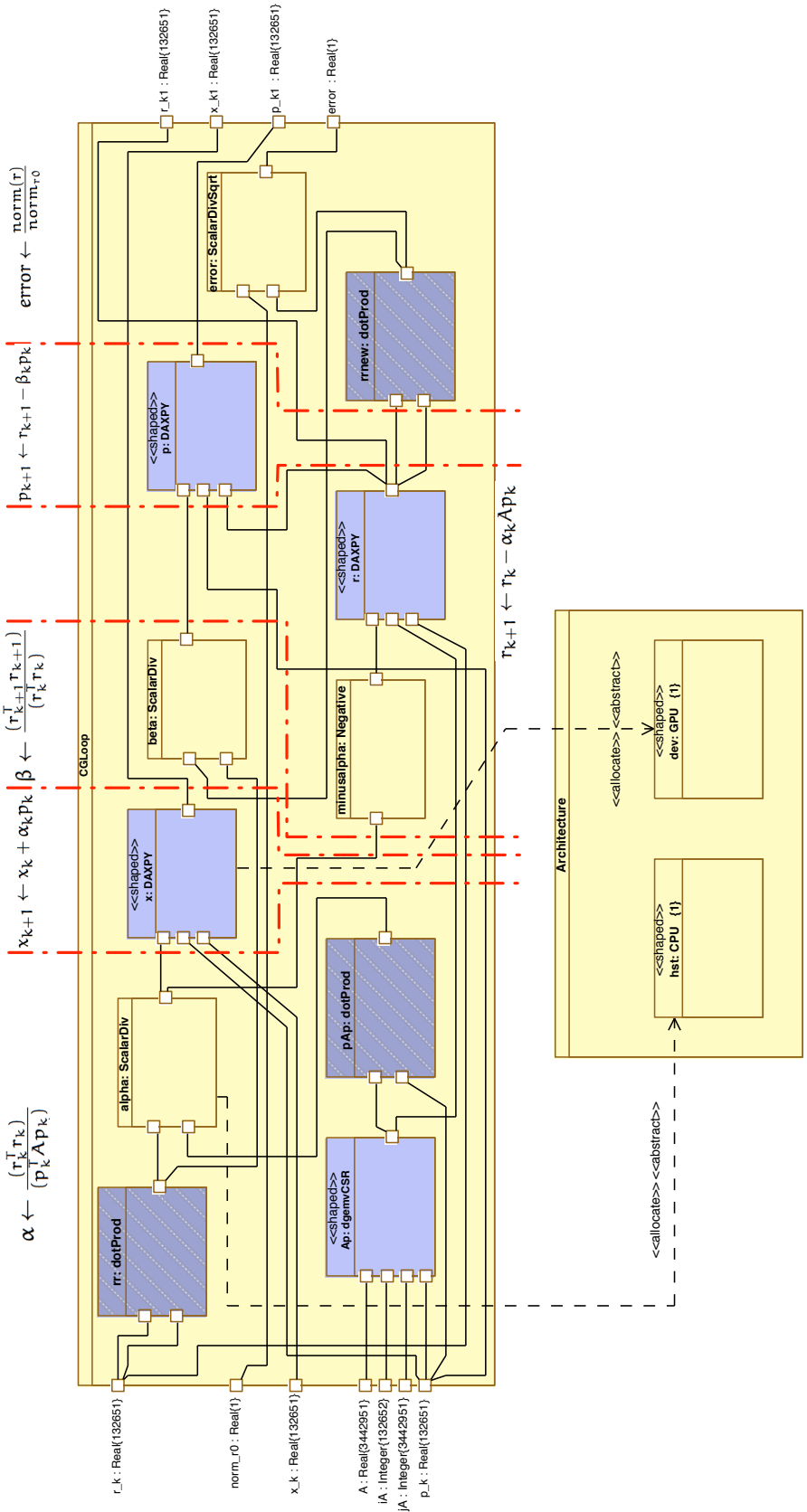


Figure 8.3: Hybrid Metamodel

PARALLEL REDUCTION ISSUE Due to parallel reductions necessary to *dot product* operations, we decided to remove the last reduction from GPU (*cf.* Figure 8.4). Actually, once the synchronization of work-items in GPUs is made at work-group level, this kind of operation must be ended on the CPU. Consequently, parallel reductions and scalar operations run on the host side. For instance, in Figure 8.4, we perform the dot product on vectors of 1000 elements. The host launches 10 work-groups containing 100 work-items. Each work-item performs the product between two corresponding elements. A special elementary task (*r:reduc*) is deployed by a special IP that does a parallel reduction and selects only the first iteration to write out the reduction result. At the end, 10 reductions are produced. The best solution, in this case, is to perform the final reduction on the host side (CPU). In such a scenario, it is necessary to take into account the vectors' sizes. Indeed, for huge vectors the number of work-items per work-group can exceed the hardware limits. We do not provide any automatic solution for this situation. However, the profiling feedback presented in Chaper 6 can help the designer to identify this problem.

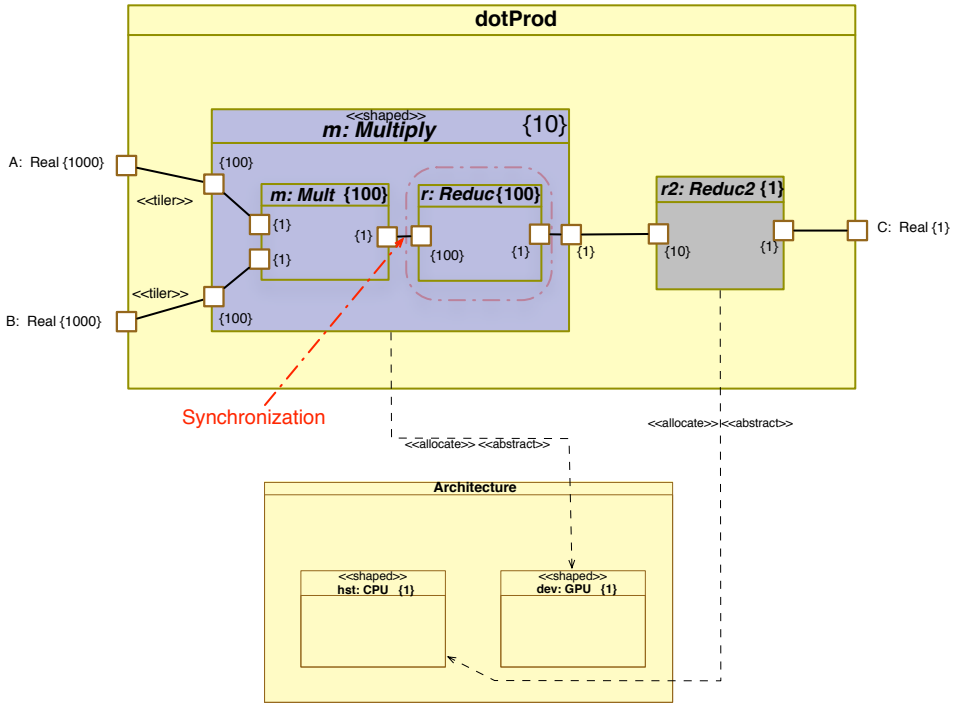


Figure 8.4: Dot Product Task

8.2.2 *Expressing the Device Multiplicity*

One of the most important aspects, already introduced in Chapter 5, is the capability of expressing device multiplicity. In Figure 8.3, the

device has multiplicity equals {1}. Specifically for our available architecture (Tesla S1070), the *«shaped» dev: GPU* can support shape value equals {1}, {2}, {3}, and {4}. As seen in Section 5.6.4, this information is undertaken by the code generation process and adapted automatically to allow sharing the task among the multiple available devices (GPUs). However, the automatic process is not suitable to dot product operations and requires small manually-written changes. Results comments are in Subsection 8.2.5.6.

8.2.2.1 *Matrix Formats and Data Sharing*

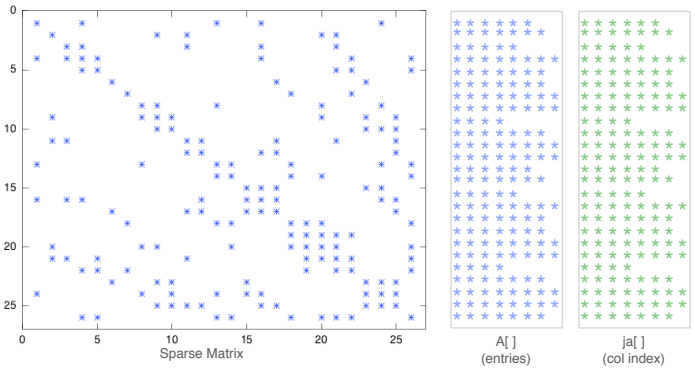


Figure 8.5: Sparse Matrix in ELLPACK-R Format

In order to equally divide a sparse matrix and to express it in ArrayOL, the CSR format should be replaced by a regular format with fixed number of rows or columns. ELLPACK or ITPACK [129] was introduced as a format to compress a sparse matrix with the purpose of solving large sparse linear systems with ITPACKV subroutines on vector computers. This format stores the sparse matrix into two arrays (as seen in Figure 8.5), one for values $A[]$, and one integer $ja[]$, to save the column index of every entry. Both arrays are of dimension $N \times MaxNNZ$ at least, where N is the number of rows and $MaxNNZ$ is the maximum number of non-zeros per row in the matrix, with the maximum number being taken over all rows. Note that the size of all rows in these compressed arrays $A[]$ and $ja[]$ is the same, because every row is padded with zeros. Therefore, ELLPACK can be considered as an approach to fit a sparse matrix in a regular data structure similar to a dense matrix. Consequently, this format is appropriate to compute operations with sparse matrices on vector architectures allowing for load balancing. In our tests, we do not change the original storage format of matrices in Code_CARMEL. Instead, we insert an intermediate function to perform the format conversion. As a result, this approach takes a relevant time and decreases the overall execution time. However, some works [144, 12] have shown good results concerning matrix-vector products into many-threaded

architectures such as GPU, when matrices are stored in regular formats such as ELLPACK. These findings lead the perspective of changing the CODE_CARMEL matrix structure when dealing with GPGPU architecture.

8.2.3 *Generated Code*

As seen in Chapter 3 a single command starts the transformation model chain UML/MARTE-to-OpenCL. The generated files for this application are summarily presented in Table 8.1.

Table 8.1: Generated Files for the CG

File	Description	# Lines
Makefile	Configuration file that is utilized by the "make" tool in order to identify the location of source files that will be used to build this application	18
oclCG_GPUApp.cpp	The host's source file that contains all host functions declarations and handles external interface, device detection, memory transfers and kernel launches	1018
ocldpcsr_KRN.cl	Source code of the kernel that is responsible for the sparse matrix-vector product	54
ocldprod_KRN.cl	Source code of the kernel that is responsible for the partial dot product	41
ocldaxpy_KRN.cl	Source code of the kernel that is responsible for the *axpy* operations	23

Object files compiled in C or Fortran are compatibles. However, some small changes to ensure the variable types equivalency are necessary.

Once having these source files compiled and linked to the object files of the main structure of CODE_CARMEL, a *software testbed* can be prepared to start the tests.

8.2.4 *Tests*

8.2.4.1 *Platform Configuration*

Although the platform defined here was already used in previous examples, we present it again in order to clarify the hardware configuration.

CPU - *AMD Opteron*: 8-core @2.4GHz and 64GB RAM.

More details about the internal aspects of GPUs can be found in Appendix A.

GPU - *Tesla S1070*: The NVIDIA® Tesla™ S1070 Computing System is a 1U rack-mount system with four Tesla T10 computing processors. This system connects to one or two host systems via one or two PCI Express cables. A Host Interface Card (HIC) is used to connect each PCI Express cable to a host. The host interface cards are compatible with both PCI Express 1x and PCI Express 2x systems.

· Four Tesla T10 GPUs

· 16.0 GB of high speed memory, configured as 4.0 GB for each GPU

Operating System - *Linux*: CentOS release 5.5.
Fortran Compiler - *F95*: Sun Studio Fortran 95.
C/C++ Compiler - *GCC*: gcc 4.1.2 20080704.
OpenCL SDK - NVIDIA GPU Computing Software Development Kit - OpenCL 3.2 Release.

8.2.4.2 *Cube Models*

For this case study we created 6 models of cubes with different mesh granularity. The cubes are made of iron and they are inserted in air. These models are created using SALOME[1] [25]. Figure 8.6 shows the representation of the cubes and their different meshes. Our tests are based on Finite Element Method (FEM) principles. We divide the domain into finite triangular subregions. For instance, cube 1 has 48 triangles and cube 6 has 49152. More we refine the triangulation, more precise will be our simulation. However, the number of triangles impacts directly the size of the problem to be solved. In Chapter 7, we introduce CODE_CARMEL as a set of modules. The module `gendof` is responsible for reading each cube model (in `.med` format) and creating a `.car` file used as input for the remaining modules.

1. SALOME is an open-source software that provides a generic platform for Pre- and Post-Processing for numerical simulation

8.2.4.3 *Formulation and Matrix Assembly*

In CODE_CARMEL, the *matrix assembly* generates a sparse symmetric positive-define matrix for each input model example. These matrices are illustrated graphically in Figures 8.7 and 8.8. The size of matrices depends on the mesh granularity and the adopted formulation, either $(\mathbf{A}, \Phi)$ or $(\mathbf{T}, \Omega)$[2]. We emphasize here two matrices. First, the smaller one (cube 1), a 26×26 matrix with 210 non-zero elements, the formulation $(\mathbf{T}, \Omega)$ gives slightly the same size. Second, the larger one (cube 6), a $1,798,336 \times 1,798,336$ matrix with $29,089,432$ non-zero elements. For the latter case, the formulation $(\mathbf{T}, \Omega)$ has fairly impact on the size of the produced matrix ($\approx 15\%$ of the $(\mathbf{A}, \Phi)$ one).

2. It is not our goal to discuss the reasons that lead to smaller or larger matrices according to the formulation.

8.2.5 *Results*

This section presents the running results using two types of charts. The first chart regards the convergence of each solver module. Conjugate gradient algorithms converge to the extremum of quadratic functions in a finite number of steps. The rate of convergence depends basically on the used algorithm [29]. CG is complete after n iterations. However, it is important to analyze the convergence. In practice, accumulated floating point roundoff error causes the residual (r_k) gradually to lose accuracy, and cancellation errors cause the search vectors to lose A-orthogonality. Nevertheless, solutions have been

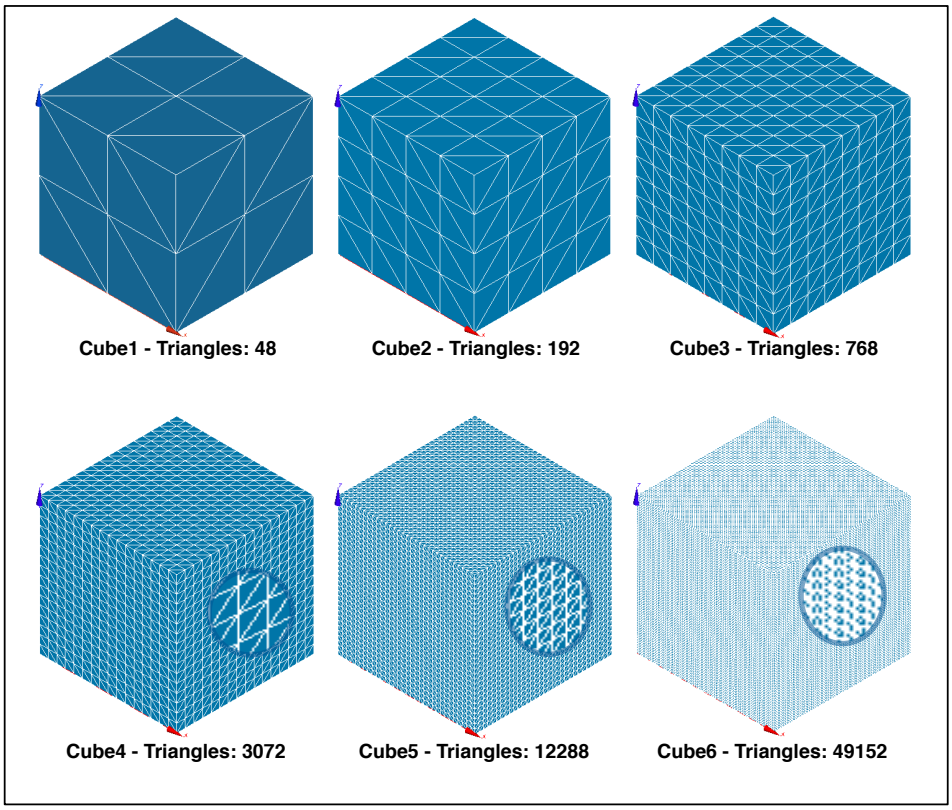
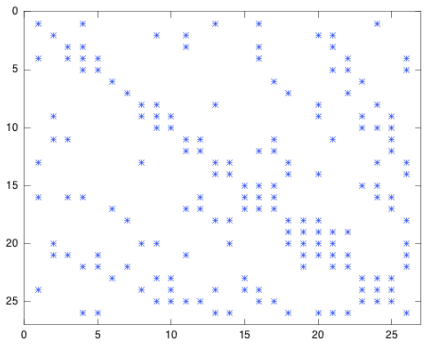
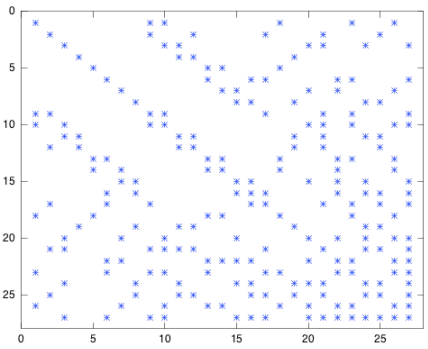


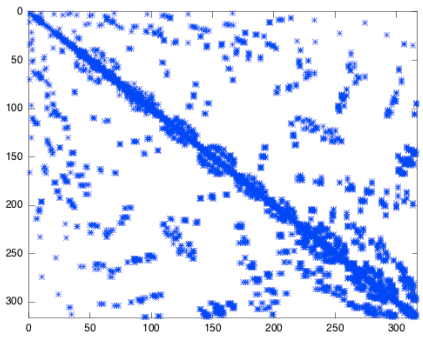
Figure 8.6: Mesh Models used in the Simulation



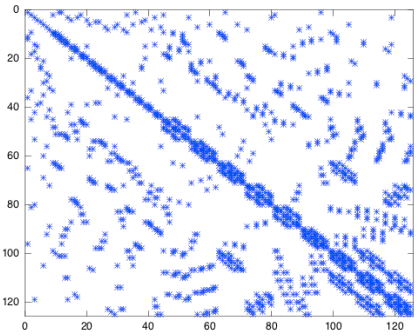
Cube 1 A-Phi: n=26, nnz=210



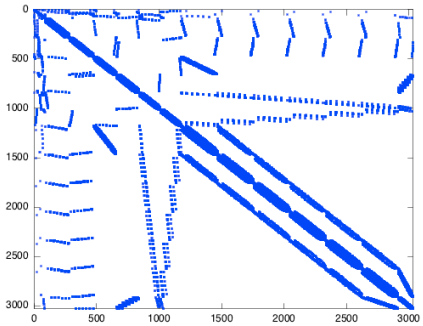
Cube 1 T-Omega: n=27, nnz=223



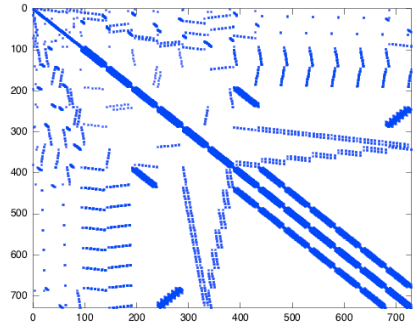
Cube 2 A-Phi: n=316, nnz=3892



Cube 2 T-Omega: n=125, nnz=1333

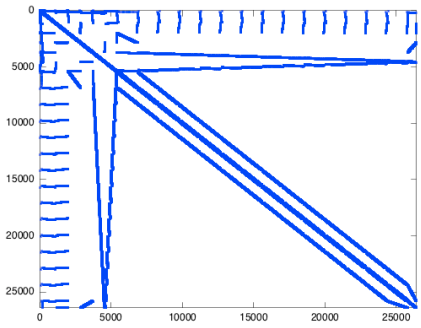


Cube 3 A-Phi: n=3032, nnz=43632

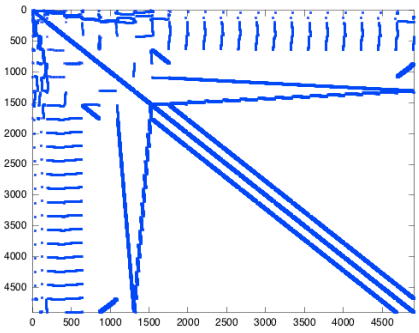


Cube 3 T-Omega: n=729, nnz=9097

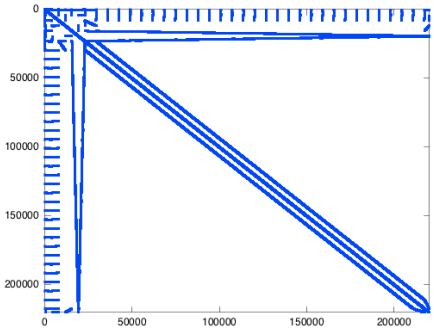
Figure 8.7: Cube 1 to 3: Sparse Matrices from Assembly Process



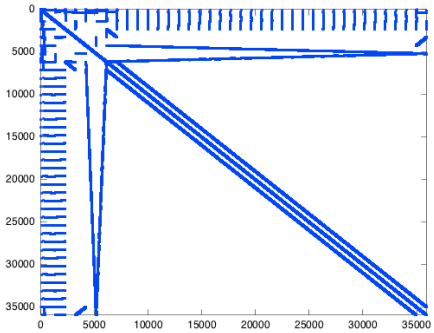
Cube 4 A-Phi: n=26416, nnz=407176



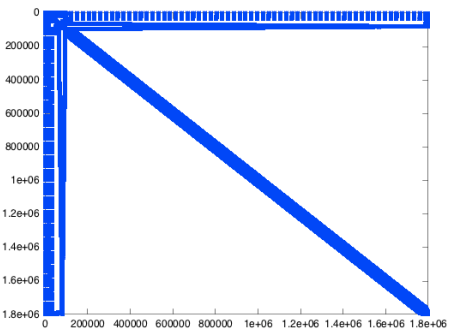
Cube 4 T-Omega: n=4913, nnz=66961



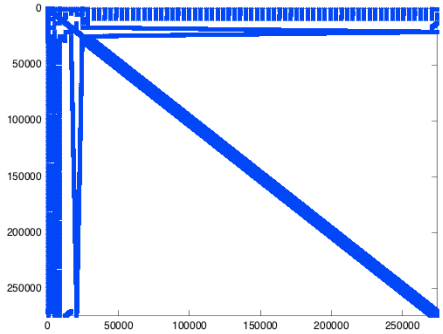
Cube 5 A-Phi: n=220256, nnz=3507000



Cube 5 T-Omega: n=35937, nnz=513313



Cube 6 A-Phi: n=1798336, nnz=29089432



Cube 6 T-Omega: n=274625, nnz=4018753

Figure 8.8: Cube 4 to 6: Sparse Matrices from Assembly Process

proposed to successfully overcome this kind of problem [68, 76]. The second type of chart shows execution times. We compare the available solutions of CODE_CARMEL on CPUs with the solution on GPUs.

8.2.5.1 *Cube 1 to 3: Convergence*

Figure 8.9 presents the convergence charts for cubes 1 to 3. The main difference in the number of steps in convergence is seen specially when we change the formulation. Another important aspect to note is the convergence between the CG implemented in Fortran90 and the one in GPU/OpenCL. They have exactly the same curve[25] .

[25] *Minimal differences in error, due to different floating point processing, exist. However, this is not visible in the charts.*

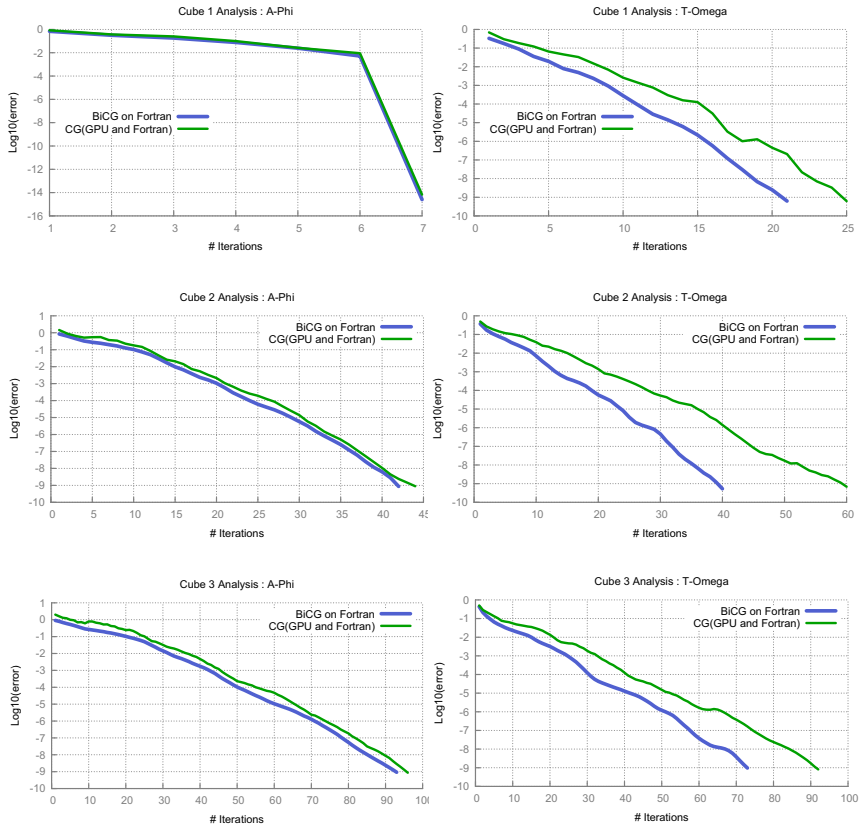


Figure 8.9: Convergence Charts for Cube 1 to 3

8.2.5.2 *Cube 4 to 6 Convergence*

The convergence charts for the cubes 4 to 6 is seen in Figure 8.10. The main points to note, is regarding the slight approximation between the two curves. Again, like the previous cubes, the curve for the both CG versions (Fortran90 and OpenCL) are the same. Moreover, as we can see in last chart in Figure 8.10, even using a more refined algorithm (the BiCGCR), it does not always assure a better convergence.

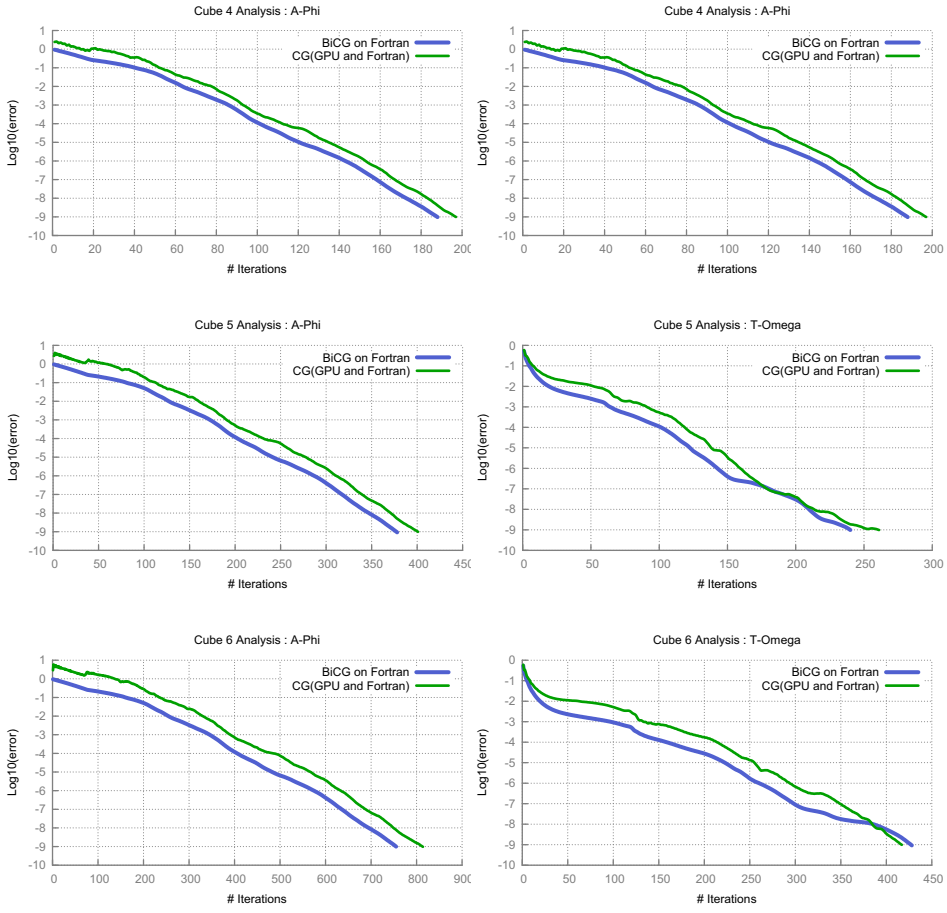


Figure 8.10: Convergence Charts for Cubes 4 to 6

8.2.5.3 *Cube 1 to 3: Time Results*

The times in all charts in this chapter are given in seconds.

Some points can be discussed regarding the results of execution times. In order to launch a kernel on devices, it is necessary to do an initial procedure every CODE_CARMEL execution. This procedure consists in compiling the kernel, initializing host variables and transferring these data to the device, then launching the kernel. Afterwards, data resulting of a kernel operation should be transferred back to host. Thus, it is normal to observe a time point of about 4 seconds in all results from solvers executed on GPU (Figures 8.11 and 8.12). This "setup" time is reduced expressively in cases where multiple iterations of CODE_CARMEL are performed. The compilation of kernels only happens once.

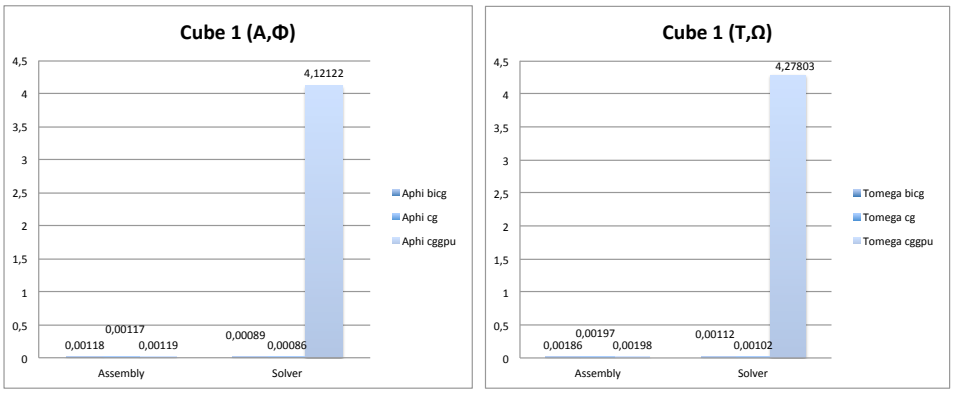


Figure 8.11: Results for Cube 1

8.2.5.4 *Cube 4 to 6: Time Results*

From cube 4, it is remarkable that a larger problem starts to decrease the overall advantage of the CPUs solution (see Figure 8.13). Indeed, in cube 5, the matrix $n \approx 220,000$ for $(\mathbf{A}, \Phi)$ formulation, we notice the bound point when the GPUs solution starts to have better performances. This is more emphasized in cube 6 ($(\mathbf{A}, \Phi)$ formulation) (Figure 8.14 when the speedup achieves about 9x for the solver and about 2x for the overall solution [1].

1. Remind that the setup and data adaptation, before launching the solver, take a significant time.

The speedup evolution is summarized in Figure 8.15. Even if the graph shows a linear increasing curve, this was not verified. Indeed, we should note the problem size of 220k degrees of freedom. From this point we achieve speedup of 1.6x, which starts to become a useful solution.

8.2.5.5 *GPU Times Analysis*

A more precise analysis for the cubes in examples 1 and 6 is shown in Table 8.2. Here, we emphasize the total time for the cube 6 $(\mathbf{A}, \Phi)$

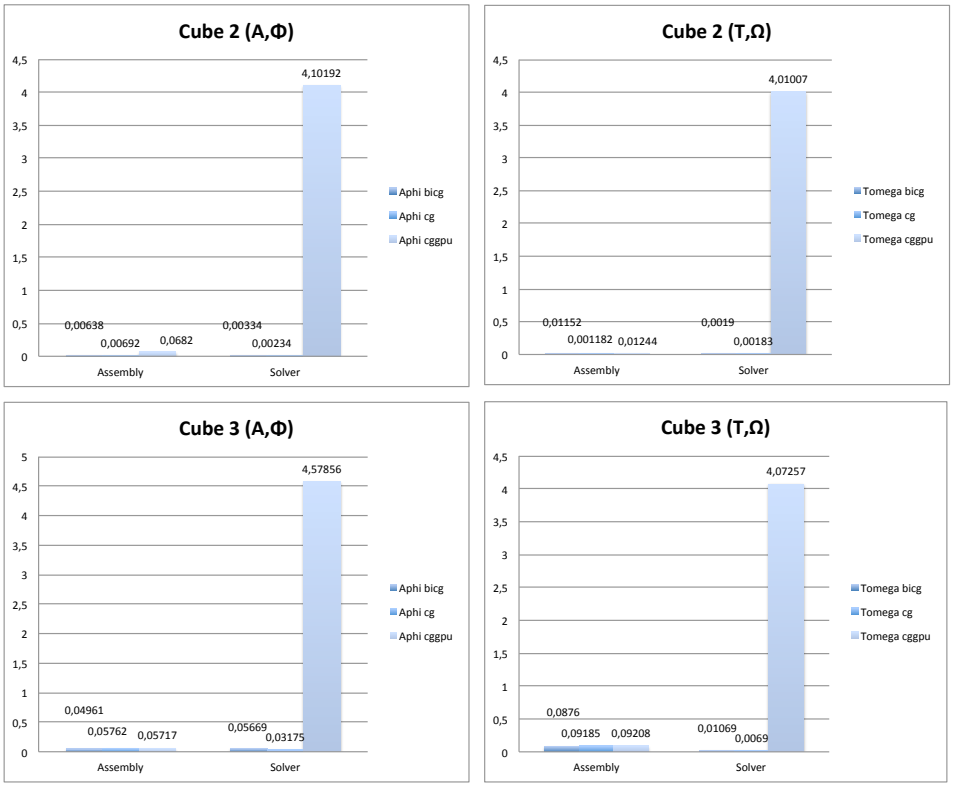


Figure 8.12: Results for Cubes 2 and 3

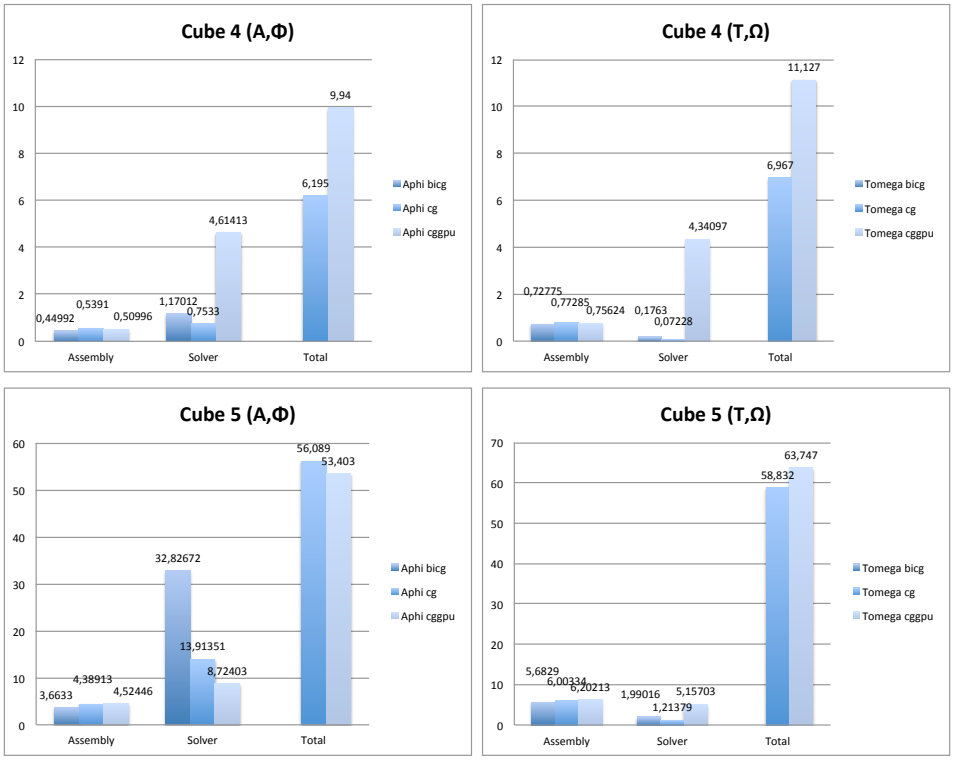


Figure 8.13: Results for Cubes 4 and 5

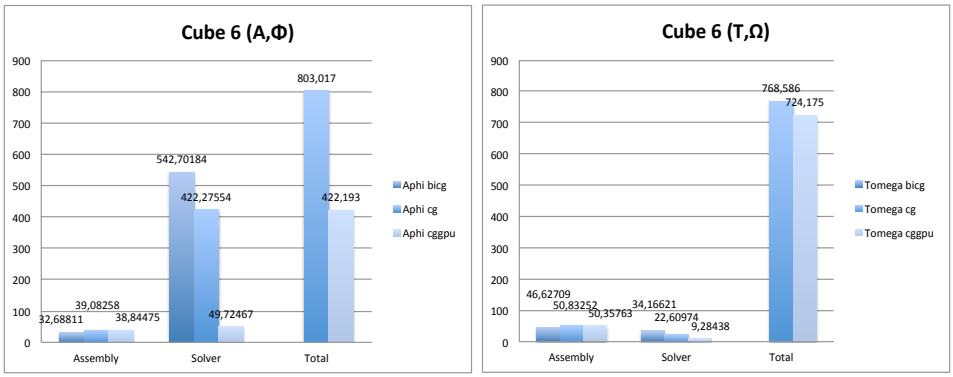


Figure 8.14: Results for Cubes 6

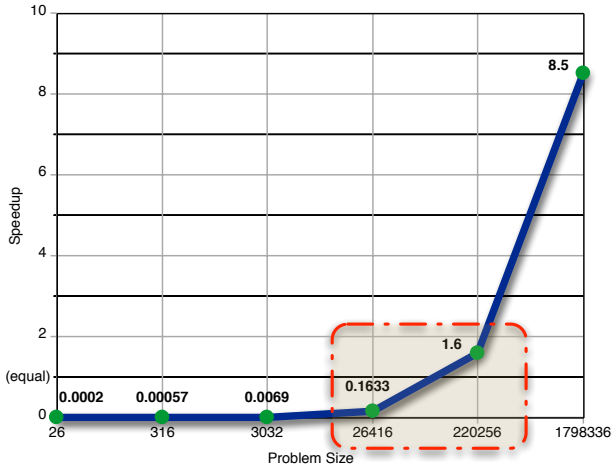


Figure 8.15: Speedup Evolution according to Problem Size

solver execution (49.72 seconds in Figure 8.14). From this execution time, 45s are due to CG loop where data transfer is responsible for about 39s. The kernel execution takes only about 8% of the total execution time. This shows how important is to decrease this difference in order to achieve better results.

Table 8.2: Cube 1 and 6: GPU Times Analysis

Case	Setup	H2D	Loop	Kernels	D2H	#Iter.	Time/Iter
Cube1: $(\mathbf{A}, \Phi)$	4.09	0.0014	0.019	0.008	0.007	7	0.00017 (CPU:0.00017)
Cube1: $(\mathbf{T}, \Omega)$	4.20	0.0012	0.068	0.027	0.025	25	0.000079 (CPU:0.000079)
Cube6: $(\mathbf{A}, \Phi)$	4.17	0.27	45.23	3.76	39.2	814	0.061 (CPU:0.52)
Cube6: $(\mathbf{T}, \Omega)$	4.12	0.047	5.09	1.08	3.28	417	0.022 (CPU:0.054)

Figure 8.16 presents the magnetic field **B** after post-processing phase of CODE_CARMEL.

8.2.5.6 *Multidevice Results*

Unfortunately, the examples shown above do not attain better performances when we increase the number of devices. In fact, the communication and kernel launch times increase about 10% for the cube 6 example. This is due to the overhead accumulated on each extra transfer and kernel launch. GPGPU is well known to be better in heavy computation algorithms. Therefore, for examples where this situation is predominant, multiple devices lead higher speedup. Nevertheless, in [41], we overlook some setup times and, thus, multiple devices

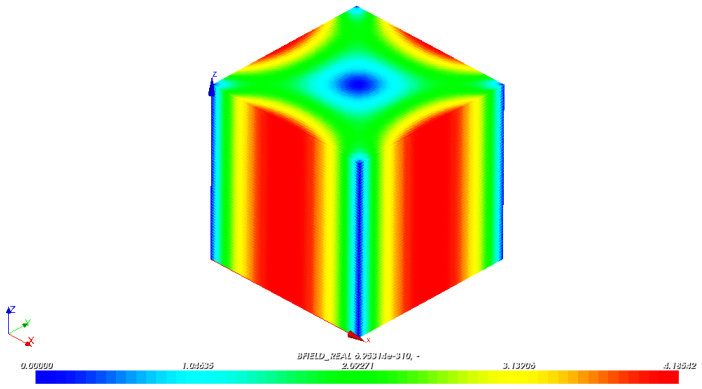


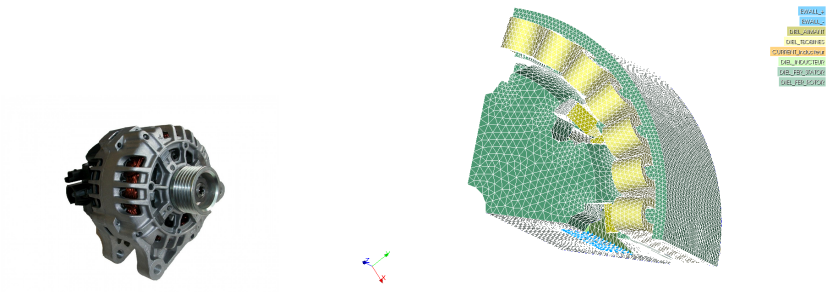
Figure 8.16: Post-processing results: **B** field for the cube

could ensure higher speedups as the number of devices increases. Table 8.3 illustrates these speedups.

Table 8.3: Multi-GPU: N=132,651, NNZ=3,442,951, tol=1e-10. *Source: [41]*

conjugate gradient	#iter	time(s)	speedup	gflops
Matlab PCG (desktop PC)	117	3.17	1	.303
OpenCL (1 GPU)	116	0.659	4.81	1.45
OpenCL (2 GPU)	116	0.461	6.87	2.07
OpenCL (4 GPU)	116	0.380	8.34	2.50

8.2.6 *Automotive Alternator Example*



(a) Real (b) Part Model

Figure 8.17: Automotive Alternator from Valeo™

The earlier cases are academical approaches and suit well to present the applicability of our methodology onto electromagnetism simulation. They point out key aspects in which Code_CARMEL takes advantage of a high-performance solver module. Now, we present its application on a real industrial example. Thus, we have chosen to

simulate and model an automotive alternator developed by Valeo™. Alternators are AC electrical generators and usually the word refers to small rotating machines driven by automotive. Subfigure 8.17a illustrates an alternator from Valeo™ and Subfigure 8.17b shows a basic model designed in SALOME of this system. We define a mesh granularity for FEM and apply Code_CARMEL on this model (.med file). From matrix assembly, this produces a linear system whose matrix A has $n = 775,689$ and $nnz = 12,502,443$.

Benchmarks for this example gives speedup of ~ 9 with relation to standard Fortran version on CPU. We had 10,000 iterations in about 2300 seconds on CPU against 250 seconds on GPU. This is a good result that shows the potential increase in speedup according to complexity of the problem[26] . For illustration, Figure 8.18 presents graphically the simulation result.

[26] *The sparse matrix produced from this simulation proportionally has more non-zero elements.*

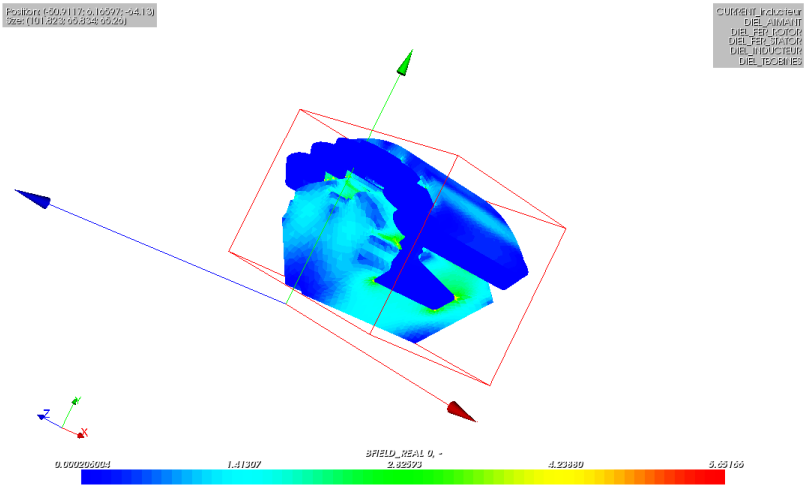


Figure 8.18: Post-processing results: **B** field for the alternator example

8.2.7 *Overall Comparisons*

We have analyzed two works [60, 26] to compare our results. Both uses sparse matrices. Georgescu et al., in [60], study the performance and feasibility of Conjugate Gradient (CG) on graphics processors. While on per iteration performance basis, the GPU is up to 13x faster than the CPU. After running the iterative refinement iterations, the overall average speedup is about 3-5x. Under the point of view of systems with multiple devices, Cevahir et al., in [26], present some benchmarks of CG on multi-GPU clusters. They performed a more complex testbed using multiple CPUs cores and multiple GPUs per node. However, comparatively their gain, it is between 5-10x with relation to sequential CPU version.

These results show that our approach is able to provide competitive performances for the CG solver on sparse matrices. Moreover, the scalability of the code is obtained by simple changes in the high-level specification models.

8.3 CONCLUSION

In summary, in this chapter, we propose to change the solver module of CODE_CARMEL by a GPGPU one. To achieve this aim, we use our methodology of development for massively parallel applications based on an MDE approach. Then, with some technical changes in makefiles and datatype compatibility, we add the generated code to CODE_CARMEL and start a testbed on some cube meshes according to Finite Element Method (FEM). Obtained results are fairly satisfactory for larger meshes in the order of tens of minutes. Furthermore, speedups with relation to sequential versions on CPUs are comparable to related works on CG and GPGPU. Another important aspect regards to multi-GPU. Although we do not have an immediate performance gain to these problems, results show that it is possible to achieve better results when we improve the ratio $\frac{\mathrm{kernel\,execution\,time}}{\mathrm{data\,transfer\,time}}$. Moreover, multi-GPU feature requires significant changes on hand coded OpenCL programs, while our approach implements this feature by simple tagged values in *HwResource* stereotypes from MARTE.

CONCLUSION AND PERSPECTIVES

CONCLUSIONS

The work discussed in this thesis report is placed in the domain of parallel programming methodology. Particularly, we address high-level specification, efficient code generation according to GPGPU programming model, and high-level modeling for algebra linear functions and numerical algorithms applied to simulation of electrical machines.

Through this report we presented in Chapter 1 different approaches concerning high-level specification for HPC and frameworks that adapt the development of applications GPGPU for several programming languages. However, these existing approaches lack one or more of the key aspects below that we consider valuable to a complete approach.

1. Compact expression of parallelism at a high-level specification.
2. Specification of architecture at higher levels.
3. Efficient code generation.
4. Specification of allocation onto multiple devices in a same host system.

Aiming at filling the gaps of these requirements, we choose Gaspard2 as a framework based on Model-Driven Engineering (MDE) to specify application and architecture. Gaspard2 uses MARTE and has several code generation branches for different targets, including OpenMP. Indeed, the OpenMP programming model also aims HPC, but it is different from our proposal by the way it addresses the problem. However, the similarities between our goals and the existing transformation modules allowed us to propose a new branch to generate a code for our intended platform: OpenCL as programming language for GPGPU. MARTE is a UML profile and it suits well to specify data and task parallelism with ArrayOL specification. Moreover, MARTE principles implement a co-design environment to distinctly create software and hardware models.

Initially, we presented the methodology from the point of view of the model designer. Two applications of different domains, matrix multiplication and video downscaling, illustrate the *modus operandi* and the potential performance gain on results and benchmarks. Afterwards, the modeling techniques seen in the examples are often referenced in the remaining of the second part of this report, when we present more details about the model compilation. Optimizations concerns are, then, discussed in order to generate an efficient code.

Regarding the simulation of electrical machines, the third part of this report presented some theory of the electromagnetism and the Finite

Element Method (FEM). This part aimed to show the feasibility and performance gains when we apply the methodology onto a module of the CODE_CARMEL that deserves parallel solutions. Then, results and benchmarks are presented as a resource to validate the approach. For the better cases we have achieved speedup of about 9x. This pragmatically demonstrates the real viability of the approach.

Moreover, during the elaboration of this work, the development and application of the methodology addressed in this thesis report led us to make some positive reflections, criticism, and perspectives in its evolution.

◦ *Development Methodology for GPGPU*

This work comprehends many points of view according to the role of the actor. From the point of view of the methodology provider, there are many aspects to take into account at model compilation time. However, these aspects are transparent to model designers. For them, we offer a methodology to specify their applications and architecture at a higher-level and generating automatically code. With this application development methodology, we give the means to academic and industrial researchers to specify software for GPGPU at a high-level abstraction. Indeed, based on our defined requirements for a development environment, our choices led to MDE as an overall solution to produce an application. Model-Driven Engineering (MDE) leads the software development to the world of the models, which is a higher-level specification. Currently, researches have shown that the hardware technology and, consequently, its programming model is evolving in about every 4 years, while new algorithms evolve every 20 years. The proposed methodology bridges this gap by offering resources to design the algorithm with low dependency on hardware. This allows for software and hardware scaling of applications. However, this solution comes with a cost. In MDE, the profile UML for MARTE was designed to assist embedded systems specification. Moreover, many additions were proposed to MARTE and, today, it is a standard to specify real-time and embedded systems with resources to express the parallelism of application and architecture in ARRAYOL. Nevertheless, even though ARRAYOL provides the resources to describe compactly data and task parallelism, it imposes constraints to the specification. This issue is specially critical for applications whose data amount varies according to the problem. However, this does not hinder the code generation. Indeed, the generated code is human readable and brings even some comments which facilitate its understanding. This allows developers to establish constants or parameters aiming for a generic problem support. Moreover, in [34], studies have been elaborated to add parameterization feature in UML models. Another drawback is concerning to code generation for libraries. In fact, our methodology is based on the GASPARD2 framework, and, therefore, we can only generate a whole

application and not individual modules. Nevertheless, some slight modification make the code useable. In other words, even with few concerns that can affect the generated code, they do not offer obstacles. Furthermore, we believe that tools in this field will increasingly be improved[27] and will provide better development environments.

Particularly in the field that we have focused on, electrical machine simulation, we consider that the methodology has fairly achieved its goal. We were capable of modeling an application and architecture, generating an efficient and functional code, and applying it directly on industrial problems. As a result, we obtained gains in performance within an existing tool of the order of 9x. These gains justify the adoption of the methodology to other modules in a complex simulation tool, such as CODE_CARMEL, or even to develop other tools regarding the potential parallelism and adaptation onto GPGPU. For instance, currently, new researches in the field of electrical machine simulation require experiments that can take more than one week to be finished (by multiple tests of the same problem). This proposal is able to reduce this time to less than one day.

[27] *Recently, new tools, such as Modelio from SoftTeam™ (`modelio.org`), have emerged as open-source tool with support to MARTE. This enforces that community investment in this area.*

◦ *Optimizations and Device Multiplicity*

In the pursuit of efficiency similar to hand coded program, we attempted to adapt good practices analysis in the model transformation chain. Proposed optimizations take into account memory access and profiling integration. The intervention of designers is still essential in some cases. For instance, using profiling integration designers are asked to refactor or modify their models manually. However, these optimizations enlighten key features according to good programming practices.

The OpenCL programming model is not exclusive for GPGPU. Indeed, even SMP computing power can be exploited by OpenCL applications. Thus, extending the number of devices in the same host environment is fairly valuable to achieve higher parallelism. However, this extension implies usually relevant code changes. The simple modification of device multiplicity facilitates the designers job whenever they need to increase scale of their applications. However, overhead in data communication hinders this feature in some applications. Notably, heavy computing applications rather than heavy data processing ones take advantage of this feature. This is of special interest to our target applications.

PERSPECTIVES

We have addressed many aspects of GPGPU and high-level specification of applications for this architecture. However, this research field is in constant evolution. Besides the previous commented issues that deserve improving, we enlighten some perspectives and future works.

GPU Clusters

We do not deal with GPU clusters. Some issues make this environment more complex. We have worked in an environment composed of one host and a few homogeneous devices. Clusters are much more complex than our local structure, and they require other programming model. In such a case, we do not have the role of the "host" from a global view. It is a fully distributed system and the communication becomes difficult to specify at higher levels and still to ensure a competitive generated code in performance. However, currently, existing GPU clusters clock an impressive 2.5 petaflops on the LINPACK scale. This peak computing rate is reached by gathering, for instance, 14,336 Xeon X5670 processors and 7,168 Nvidia Tesla M2050 GPGPUs. The programming model for this situation requires distributed memory solutions. MPI is usually the programming model adopted by developers in cluster environments. Given the popularity of GPU clusters in academic and industrial areas, as a perspective, we suggest enabling the specification of clusters at architecture level and the integration with MPI libraries. In such a situation, the synchronization, communication, and the distributed memory are the most issues to be overcome. However, an in-depth study can generalize our localized solution for multi-devices.

Control on Code Generation

Although the code generation process is based on a layered transformation chain, this process is not parameterized. No control structure exists and, thus, designers cannot enable or disable some features at generation time. We suggest integrating a generic metamodel that allows us implementing controlled model compilation. This is particularly interesting when specifying, for instance, optimization levels or defining the generated code as either program or function library. Gaspard2 does not provide a link to interact directly with its transformation engine. This is transparent to model designers. We propose to extend the works [34] already started in this field to fill these requirements as a generic solution for all available chains.

Algorithms for Simulation of Electromagnetism Phenomena and GPGPU

We are convinced that it is not totally possible to separate software from hardware when specifying them. Indeed, we have even provided the profiling integration aiming at obtaining feedback directly on the application model at runtime. However, we can improve it more. Recently, a study of the fundamental obstacles to accelerate the Conjugate Gradient (CG) method on GPUs is presented in [44] where several techniques are proposed to enhance its performance over the generic

algorithm and that does not depend on the matrix sparsity pattern. An advanced study to enable designers thinking about modifying their algorithms aiming a specific target platform (e. g. GPU) is a relevant issue in parallel programming development.

Part IV

APPENDIX

A

HIGH PERFORMANCE COMPUTING

A.1 HISTORY

The term High Performance Computing (HPC) was originally used to describe powerful, number-crunching supercomputers. As the range of applications for HPC has grown, however, the definition has evolved to include systems with any combination of accelerated computing capacity, superior data throughput, and the ability to aggregate substantial distributed computing power.

In the last two decades, the HPC has been revolutionized by approaches based on cluster technology including high performance nodes. Nevertheless, before clusters, the HPC used to be made by large centralized systems. In this configuration, users shared centralized resources and this allowed to decrease overall costs.

However, researchers realized that these large systems could not meet the increasing computational demands over time. In other words, even if the machines become faster, the time available for HPC researchers reduced due to increasing number of users. In addition, every time a new machine with a different architecture was available in the market, programmers were forced to rewrite their code to adjust them to the architecture of this machine.

Computers had a performance increasing at a pace even faster than the centralized HPC systems. In 1993, recognizing the potential for a new innovation, Tom Sterling and Don Becker tried to combine computers with open source software to create a "personal" HPC system as an alternative to large centralized systems [71]. The idea was to put computational power with the same level available in HPC system directly to individual user.

The first system, using 16 486DX4100 cards, Ethernet 10Mbps with channel aggregation and Linux, proved to be a big gap in terms of price and performance compared to the HPC systems at the time. With the cluster "Beowulf" [15], as this system was called, researchers had individual performance levels of HPC at their fingertips for a lower price. Still could use centralized HPC systems for larger jobs, though the Beowulf cluster could handle all the traditional HPC applications.

From that point, there was an explosion in the use of clusters. Nowadays, more than 80% of the world's fastest systems in the Top

[28] www.top500.org

500 list[28] are classified as clusters. Why HPC clusters have been so successful? There are several answers, but some basic reasons are:

- **Price/Performance Relationship:** HPC clusters offer an interesting price/performance factor with respect to former centralized systems.
- **HPC for a single user:** one of the aims of clusters was to provide computational power to more and more users, and thus, disseminate the HPC practice.
- **Cluster Scaling:** it is possible to create a minimal cluster, suitable to the application requirements or create a larger system able to handle more complex applications.
- **Open source Resources:** Linux and required tools used in clusters Beowulf are usually open source softwares. Researchers do not have to buy expensive software licenses.
- **Open Standards:** the key of the success of HPC clusters is the use of systems with open standards. They use standards such as x86 processors, Ethernet, Message Parsing Interface (MPI). Hence, in the most cases it is not necessary to re-write code when we have to migrate or release an application.

Even if the performance of several machines working in parallel is very interesting, researchers noticed the importance of increasing the local performance per machine itself. With the constant evolution of hardware, specially Central Processing Unit (CPU) and graphics chipsets, i. e., Graphics Processor Unit (GPU), and the need for more processing power, various researchers began to think about using GPUs for processing generic applications and not just for specific-domain applications. GPU is a dedicated processor, normally used for graphics calculations, geometric operations and floating point operations. When we use GPUs to implement other applications beyond graphics ones, we call it General-Purpose computation on Graphics Processing Unit (GPGPU). In Section A.4, we present the state of art on GPGPU.

A.2 EXISTING APPROACHES

There is no only one way to compute in parallel. Actually, Michael J. Flynn [54] proposed, in 1972, the following classification based on the magnitude of interactions of the instructions and data streams:

1. **SISD**: Single Instruction, Single Data stream can be considered as the zero-parallel machine. No parallelism is exploited in either the instruction nor the data streams. This comprehends the traditional single-core CPU.
2. **SIMD**: Single Instruction, Multiple Data stream is when a single machine instruction is executed on mulitiple data streams. This type of parallelism is called Data Parallel and is found on GPGPUs and array processors.

Indeed, on GPGPU we have the so-called Single Program, Multiple Data (SPMD). In this case, inside a program, multiple threads can follow different paths with different instructions.

3. **MIMD**: Multiple Instruction, Multiple Data stream can be exemplified by clusters. Each node is independent and has its own memory (distributed memory).
4. **MISD**: Multiple Instruction, Single Data stream. This is a type of parallel computing architecture where many functional units perform different operations on the same data. In principle, this is not useful in practice. However, pipeline[29] architectures belong to this type, though a purist might say that the data is different after processing by each stage in the pipeline. If we take into account the purist's thought, we can consider this type of architecture does not have any practical use in HPC. But they are always interesting when we have fault-tolerant computers executing the same instructions redundantly in order to detect and mask errors, in a manner known as task replication, may be considered to belong to this type.

[29] *Pipeline is a set of data processing elements connected in series, so that the output of one element is the input of the next one.*

Multiple Instruction, Multiple Data (MIMD) is the most flexible and can execute any of the other modes. For this reason, it is the most commonly used parallel architecture. Many HPC applications are Data Parallel and, thus, they can be accelerated with GPGPUs. Nevertheless, the GPGPU architecture, has a specific design for data parallel processing, consequently it often executes data parallel operations much faster than a standard MIMD architecture, such as a multi-core processor.

A.2.1 *Architecture*

In this subsection, we emphasize some of parallel architectures in order to have a basis knowledge of existing technologies and their relation with GPGPU.

A.2.1.1 *SIMD*

The first use of Single Instruction, Multiple Data (SIMD) architectures was in array supercomputers and it was disseminated by Cray [31] in 70's. Modern high-performance hardware architectures are characterized by two distinct features: parallelism through many cores/execution units, and a SIMD-way of execution inside each core. Today's CPUs are highly parallel processors with different levels of parallelism. We find parallelism everywhere from the parallel execution units in a CPU core, up to the SIMD (Single Instruction, Multiple Data) instruction set and the parallel execution of multiple threads. Intel provides Streaming SIMD Extensions (SSE) [136] instruction set, which is an extension to the x86 architecture and using it is called vectorization. In Computer Science vectorization is the process of converting an algorithm from a scalar implementation, which does an operation one pair of operands at a time, to a vector process where a single instruction

can refer to a vector (series of adjacent values). SIMD instructions operate on multiple data elements in one instruction and make use of the 128-bit SIMD floating-point registers.

Programmers can exploit vectorization to speedup certain parts of their code. Writing vectorized code can take additional time but is mostly worth the effort, because the performance increase may be substantial. One major research topic in computer science is the search for methods of automatic vectorization: seeking methods that would allow a compiler to convert scalar algorithms into vectorized algorithms without human assistance. Indeed, some compilers are able to generate automatically SSE instructions. So, if vectorization is enabled, the compiler will use the extra unused space in the SIMD registers to perform additional operations in a single instruction.

Regarding GPUs, they are often wide SIMD implementations, capable of branches, loads, and stores on 128 or 256 bits at a time.

A.2.1.2 *Multi-core and SMP*

Multi-processing simply means putting multiple processors in one system. Symmetric Multi-Processing (SMP) implies that all of these processors are identical, also known as a homogeneous system. SMP systems have been around in the x86 world for a long time, and there are software systems that take advantage of SMP well.

From a technical standpoint, the difference between multi-core and SMP is positive. In an SMP system, each processor plugs into a different socket, and multiple processors are connected through some kind of bus. In a multi-core processor, the "core" logic of a processor is replicated multiple times on the same chip. Multiple cores may share data through some on chip logic or shared caches. Multiple cores are presented to applications at the OS level exactly the same way as multiple processors in an SMP system. Furthermore, you can mix the two together, e.g. by having an 8-core system with two processors, each containing four cores.

For the current development of applications, SMP is a choice, that some developers chose to (and still choose to) take advantage of. Multi-core is a given, that all applications must react to in order to continue to scale up performance. The reason is simple: physical barriers relating to power and heat being reached in terms of what can be done to increase the performance of a serially executing processor. The SMP era is ending. Multi-core (or many-core if we refer to GPUs) is the answer to bridge this barrier.

A.2.1.3 *Cluster*

In computers available to the general computing community, the main memory is usually shared between all processing elements (shared memory). The idea of multicomputers, often called distributed

computers, is the use of multiple computers to work on the same application. A distributed computer is a computing system in which the computers (processing elements with their own memory) are connected by a network. If the computers are located in a Local Area Network (LAN) they are called a computing cluster. For wider extensions where machines are connected in a Wide Area Network (WAN) such as the Internet, this is called grid computing.

Unlike SMPs, computing clusters are made of collections of *zero-sharing* workstations and (even SMP) servers (nodes), with different speeds and memory sizes, possibly from different generations. In such systems the user is responsible to allocate the processes to the nodes and to manage the cluster resources.

A.2.1.4 *Coprocessors*

General-purpose processors are good for overall applications and not so high-performant, however, for specific applications. More and more processors have emerged with dedicated support to interface co-processors. For instance, the ARM [55] processor provides an interface to 16 coprocessors. Another system, the NVIDIA's Tegra 2 [110] (as seen in Figure A.1), has eight purpose optimized processors. Each of these processors is power managed at a global level through software and locally through hardware mechanisms. For instance, Tegra 2 offers a video decode processor that is used to decode video streams from files that are either played from the local disk or streamed off the network. The idea behind this coprocessor is to meet the today's needs in mobile web navigation. Over 80% of today's top Web sites include video, most of which is Flash-based video content. The dedicated video processor handles all three Flash video formats: H.264, Sorenson and VP6-E. As a result, Flash video runs on NVIDIA Tegra at full frame rates and consumes very low power. Other mobile solutions that use general-purpose CPUs for Flash video deliver stuttering images and drain battery life. The decode processor is based on several generations of NVIDIA hardware decode expertise and is able to deliver flawless 1080p video playback while consuming less than 400 mW of power.

GPUs are seen as coprocessors by CPUs. Actually, not all applications suit well to the increased parallelism offered by GPUs. However, CPUs can dispatch specific tasks to GPUs aiming to achieve better overall performances.

A.2.2 *Parallel Programming*

Parallel programming has been always a challenge for developers and programming languages designers. For many programmers, sequential programming is largely less complex than parallel programming. The main challenge can be summarized in one word:

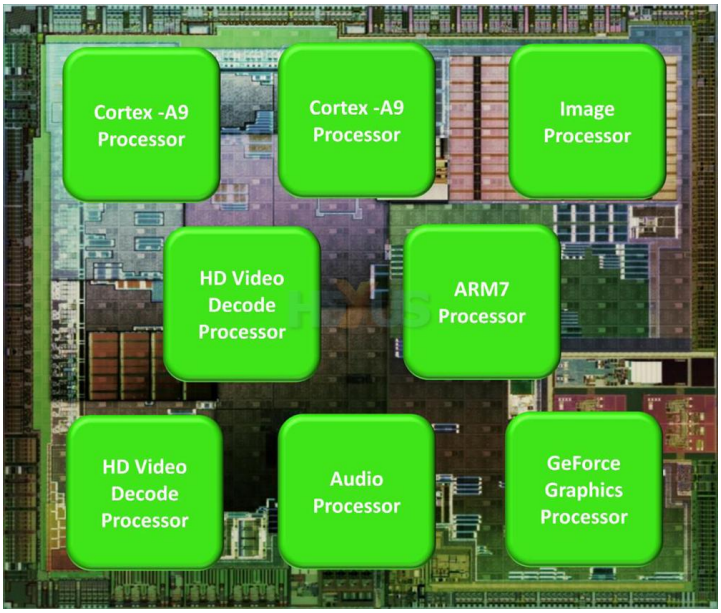


Figure A.1: NVIDIA's Tegra 2 Architecture. *Courtesy: NVIDIA*™

concurrency. Concurrency is the notion of multiple things happening at the same time. With the proliferation of multicore CPUs and the realization that the number of cores in each processor will only increase, software developers need new ways to take advantage of them. Although operating systems offers support to running multiple programs in parallel, most of those programs run in the background and perform tasks that require little continuous processor time. It is the current foreground application that both captures the user's attention and keeps the computer busy. If an application has a lot of work to do but keeps only a fraction of the available cores occupied, those extra processing resources are wasted.

The traditional way for an application to use multiple cores is to create multiple threads. However, as the number of cores increases, there are problems with threaded solutions. The biggest problem is that threaded code does not scale very well to arbitrary numbers of cores. We can not create as many threads as there are cores and expect a program to run well. What we would need to know is the number of cores that can be used efficiently, which is a challenging thing for an application to compute on its own. Even if we manage to get the numbers correct, there is still the challenge of programming for so many threads, of making them run efficiently, and of keeping them from interfering with one another.

GPGPU offers a multi-threaded environment. However, even if it is possible to re-fit the threads distribution among its cores aiming to achieve better performances, it hides the internal scheduling process. Details about how GPUs deal with threads is introduced in Section A.4.

The remaining of this subsection deal with two major approaches that envisage on one hand, OpenMP for shared memory systems, on the other hand, MPI distributed memory systems. We emphasize those approaches due to their large use in industry and academia and, moreover, they have important relationship with GPUs in particular when we integrates SMP, multi-core, and cluster computing in a same complex system.

A.2.2.1 *Shared Memory*

Once the vendors had the technology to build moderately priced SMPs, they needed to ensure that their compute power could be exploited by individual applications. Compilers had always been responsible for adapting a program to make best use of a machine's internal parallelism. Unfortunately, it is very hard for them to do so for a computer with multiple processors or cores. The reason is that the compilers must then identify independent streams of instructions that can be executed in parallel. Techniques to extract such instruction streams from a sequential program do exist, and, for very simple programs, it may be worthwhile trying out a compiler's automatic (shared-memory) parallelization options. However, the compiler often does not have enough information to decide whether it is possible to split up a program in this way. It also can not make large-scale changes to code, such as replacing an algorithm that is not suitable for parallelization. Thus, most of the time the compiler will need some help from the user.

OpenMP [27], a portable programming interface for shared memory parallel computers, was adopted as an informal standard in 1997 by computer scientists who wanted a unified model on which to base programs for shared memory systems. OpenMP widely disseminated, offers significant advantages over hand-threading. OpenMP is not a full language itself, it consists of directives and pragmas for a shared-memory application programming interface whose features are based on prior efforts to facilitate shared-memory parallel programming. Rather than an officially sanctioned standard, it is an agreement reached between the members of the Architecture Review Board (ARB) [121], who share an interest in a portable, user-friendly, and efficient approach to shared-memory parallel programming. OpenMP is intended to be suitable for implementation on a broad range of SMP architectures. As multicore machines and multithreading processors spread in the marketplace, it might be increasingly used to create programs for uniprocessor computers also.

Approaches such as OpenMP to GPGPU proposed in [97] intend to bridge two gaps: first, translating existing application's code in order to make it run on GPUs; second, an integration approach taking advantage of multi-core and many-core devices (GPUs) at the same time.

A.2.2.2 *Distributed Memory*

The world of parallel multiple instruction, multiple data, or MIMD, computers is, for the most part, divided into distributed-memory and shared-memory systems (earlier already discussed). From a programmer's point of view, a distributed-memory system consists of a collection of core-memory pairs connected by a network, and the memory associated with a core is directly accessible only to that core, as seen in Figure A.2. *Message-passing* is well suitable to this kind of system. The implementation of message-passing that we will be presenting is called MPI. Like OpenMP, MPI is not a new programming language. It defines a library of functions that can be called from C, C++, and Fortran programs (target languages also aimed by OpenMP).

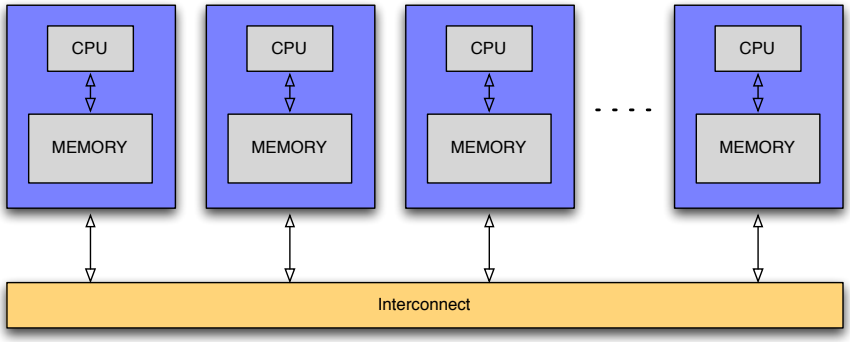


Figure A.2: Nodes in a Distributed Memory System

[30] *A GPU cluster is a computer cluster in which each node is equipped with a GPU. As an example, the GPU cluster at STFC Daresbury Laboratory has its configuration available in* `http://www.cse.scitech.ac.uk/disco/cseht/cseht.shtml`. *Many GPU clusters are part of the Top500 list.*

Many of the HPC applications have been implemented using MPI. Specially in a GPU cluster environment[30] , the simplest way to start building an MPI application is to integrate the vendor's compiler (such as NVIDIA's nvcc) for compiling everything. Aspects of programming languages for GPUs are similar to both MPI and OpenMP in that the programmer manages the parallel code constructs, although OpenMP compilers do more of the automation in managing parallel execution. Several ongoing research efforts aim at adding more automation of parallelism management and performance optimization to languages used with GPUs.

A.3 MASSIVELY PARALLEL PROCESSING (MPP)

Michio Kaku, in his book [87], states that by 2020 or soon afterward, Moore's law [105] will gradually cease to hold true and Silicon Valley may slowly turn into a rust belt unless a replacement technology is found. Transistors will be so small that quantum theory or atomic physics takes over and electrons leak out of the wires. For example, the thinnest layer inside your computer will be about five atoms across. At that point, according to the laws of physics, the quantum

theory takes over. The Heisenberg uncertainty principle states that you cannot know both the position and velocity of any particle. This may sound counterintuitive, but at the atomic level you simply cannot know where the electron is, so it can never be confined precisely in an ultra-thin wire or layer and it necessarily leaks out, causing the circuit to short-circuit. According to the laws of physics, eventually the Age of Silicon will come to a close, as we enter the Post-Silicon Era.

Nevertheless, even though Moore's law is no more valid, technologies based on parallelism such as Massively Parallel Processing (MPP) can keep better performance in applications and the global speed of processing will be continuously increased.

A.4 GENERAL-PURPOSE COMPUTING ON GRAPHICS PROCESSING UNIT (GPGPU)

For more than two decades computer applications had rapid performance increases driven by microprocessor based on a single CPU, such as those in the Intel® and AMD® families. GPGPU is a fairly recent trend in computer engineering research. GPUs are co-processors that have been widely optimized for computer graphics processing. Computer graphics processing is a field dominated by data parallel operations, particularly linear algebra matrix operations.

In order to understand the main idea behind GPGPU, we present the hardware details of most commonly used GPU.

A.4.1 *Architecture of a Modern GPU*

Although many other simpler and cheaper graphics cards are able to provide GPGPU, we have chosen the dedicated Tesla S1070 GPU computing system as architecture to depict in this subsection. This choice is due to two aspects: first, the S1070 is the hardware used in almost all of our experimental tests; second, Tesla is the first system released by NVidia as a true system dedicated to GPGPU and it has the most features available in modern GPUs.

The Tesla S1070 GPU (*cf.* Figure A.3) computing system is based on the T10 GPU from NVIDIA. It can be connected to a single host system via two PCI Express connections to that host, or connected to two separate host systems via one PCI Express connection to each host. Each NVIDIA switch and corresponding PCI Express cable connects to two of the four GPUs in the Tesla S1070. If only one PCI Express cable is connected to the Tesla S1070, only two of the GPUs will be used. To connect all four GPUs in a Tesla S1070 to a single host system, the host must have two available PCI Express slots and be configured with two cables.

The Tesla S1070 is composed of 4 T10 GPU with 4GB DRAM each one making a total of 16GB. The massively parallel T10 chip (*cf.* Fig-

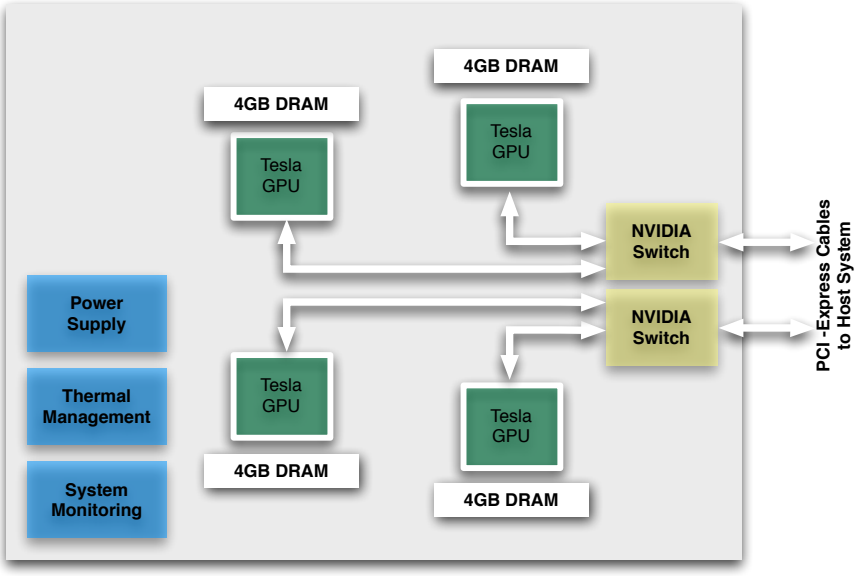


Figure A.3: Tesla S1070 Card Architecture Overview

ure A.4) has 30 Streaming Multiprocessors (SM) (10 Thread Processors x 3 SM) with clock frequency of 1.45GHz. Each Streaming Multiprocessor contains 8 Streaming Processors. This makes a total of 240 Streaming Processors (SP), i. e. 240 cores where threads can be allocated. With 240 SPs, the T10 exceeds 1 teraflops in single precision and 87 gigaflops in double precision. Because each SP is massively threaded, it can run thousands of threads per application. A usual application typically runs 5000-12,000 threads simultaneously on this chip. As an comparative example, Intel CPUs support 2 or 4 threads, depending on the machine model, per core. The T10 supports up to 1024 threads per SM and up to about 30,000 threads for the chip. Thus, the level of parallelism supported by GPU hardware is increasing quickly. It is very important to strive for such levels of parallelism when developing CPU parallel computing applications [91]. Further, in Figure A.4, each SM owns instructions and data caches, and a small size (about 16KB) memory shared among SP.

In summary, the T10 GPU belonging to S1070 has the following features:

- 30 Streaming Multiprocessors @ 1.45 GHz with 4/1 GB RAM.
- 1 TFLOPS single precision (IEEE 754 floating point).
- 87 GFLOPS double precision.
- 8 SP Thread Processors.
- 16K 32-bit registers.
- 2 SFU Special Function Units.
- 1 Double Precision Unit (DP).
- Fused multiply-add.
- Scalar register-based ISA.

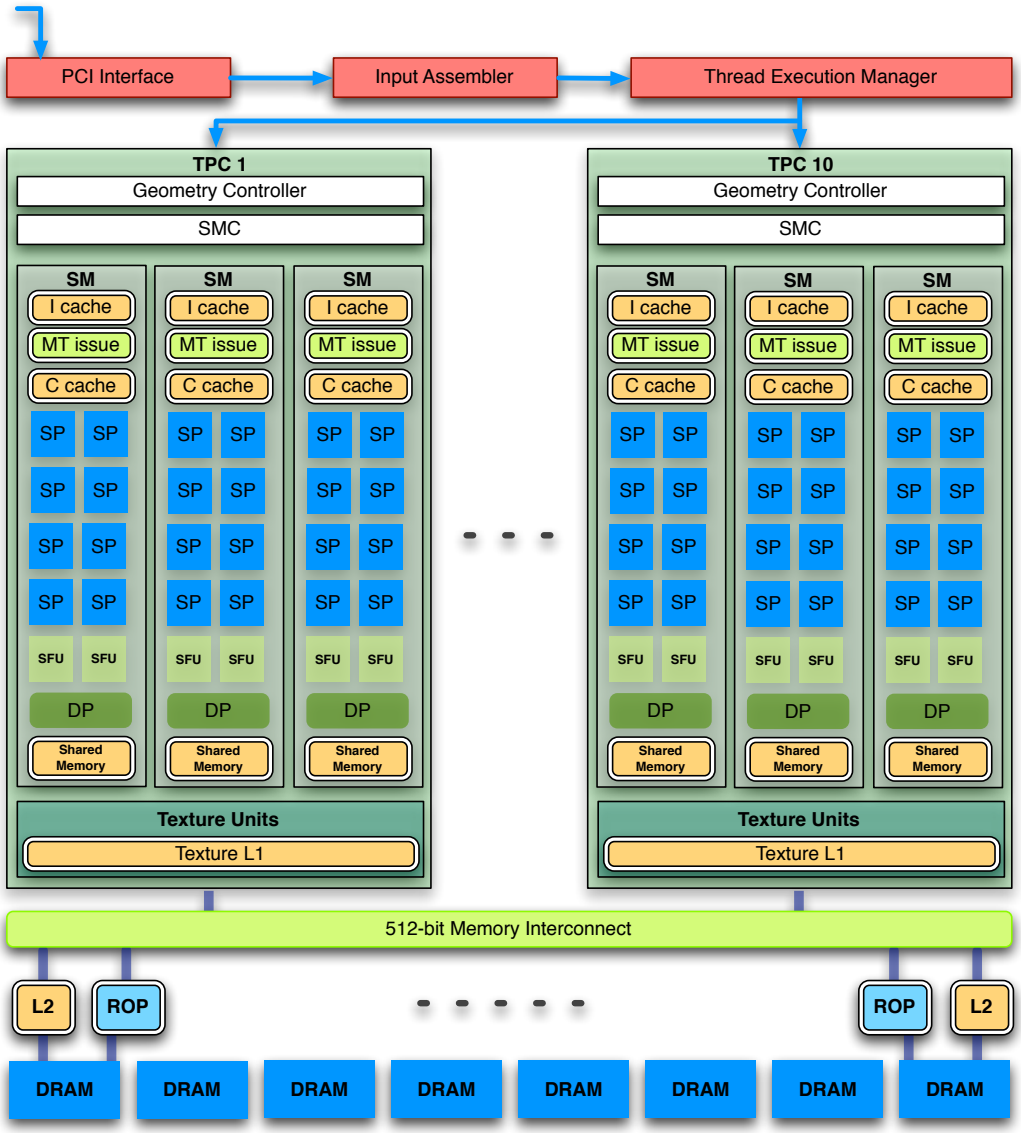


Figure A.4: T10 GPU Architecture

- Multithreaded Instruction Unit.
- 1024 threads, hardware multithreaded.
- Independent thread execution.
- Hardware thread scheduling.
- 16 KB Shared Memory.
- Concurrent threads share data.
- Low latency load/store.

A.4.2 *OpenCL™ as Programming Model for MPP*

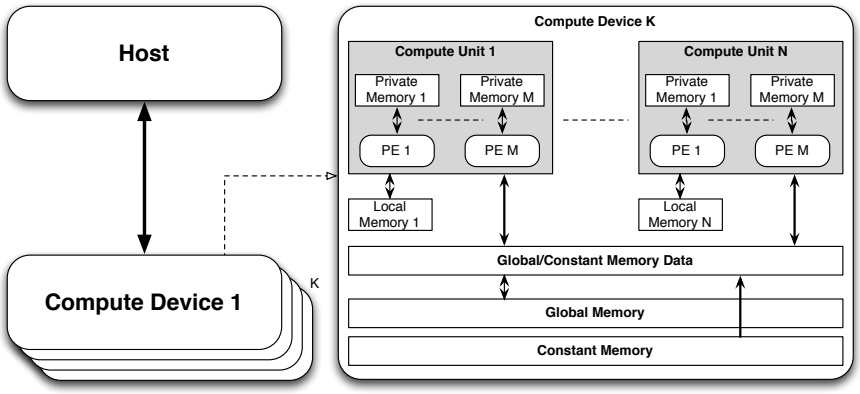


Figure A.5: OpenCL Platform and Memory Model. *Adapted from OpenCL Specification* [106].

Originally, was proposed by Apple, and then turned over to the Khronos Group [10]. OpenCL is a standard for parallel computing consisting of a language, API, libraries and a runtime system.

The diagram in Figure A.6 represents the OpenCL specification as a class diagram using the UML[31] notation. The diagram shows both nodes and edges which are classes and their relationships. As a simplification it shows only classes, and no attributes or operations. As for relationships it shows aggregations (annotated with a solid diamond), associations (no annotation), and inheritance (annotated with a triangular arrowhead). The cardinality of a relationship is shown on each end of it. A cardinality of "*" represents "many", a cardinality of "1" represents "one and only one" and a cardinality of "0:1" represents "optionally one". The navigability of a relationship is shown using a open regular arrowhead. Enlightening some key points of this diagram, the three comments show that a "Program" needs the "DeviceID" when it is built, a "Kernel" needs the "DeviceID" when it is launche, and yet, a "Kernel" takes "MemObjects" as arguments.

[31] *We introduces UML as part of Model Driven Engineering in Chapter B.*

To describe the core ideas behind OpenCL, we will use a hierarchy of models:

- Platform Model

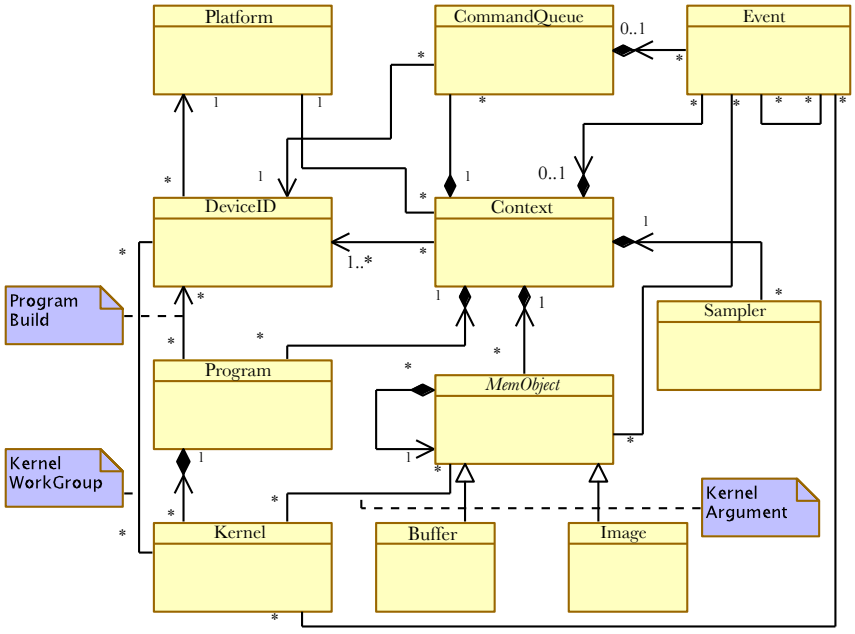


Figure A.6: OpenCL UML Class Diagram

- Execution Model
- Memory Model
- Programming Model

A.4.2.1 *OpenCL Platform Model*

The Platform model for OpenCL is defined also in Figure A.5. The model consists of a host connected to one or more OpenCL devices. An OpenCL device is divided into one or more compute units (CUs) which are further divided into one or more processing elements (PEs). Computations on a device occur within the processing elements. An OpenCL application runs on a host according to the models native to the host platform. The OpenCL application submits commands from the host to execute computations on the processing elements within a device. The processing elements within a compute unit execute a single stream of instructions as SIMD units (execute in lockstep with a single stream of instructions) or as SPMD units (each PE maintains its own program counter).

Platform Mixed Version Support. *OpenCL is designed to support devices with different capabilities under a single platform. This includes devices which conform to different versions of the OpenCL specification. There are three important version identifiers to consider for an OpenCL system: the platform version, the version of a device, and the version(s) of the OpenCL C language supported on a device.*

A.4.2.2 *OpenCL Execution Model, Command Queues and Kernels*

OpenCL provides both task and data parallelism. Data movements are coordinated via com- mand queues which provide a general means of specifying inter-task relationships and task execution order that obeys the dependencies in the computation. OpenCL may execute several tasks in parallel, if they are not dependent. Tasks are com-

prised by data-parallel kernels which, similarly to shaders, apply a single function to a range of elements in parallel. Only restricted synchronization and communication is allowed during kernel execution. OpenCL kernels execute over an 1, 2 or 3 dimensional index space. For every element in the index space a work-item is executed. As with shaders, all work-items execute the same program (kernel) and their execution may diverge due to branching depending on the data or their index. The index space is regularly divided into work-groups. See Figure 2.2. Each work- item is labelled by a global ID, unique through the whole kernel index space and a local ID, unique in its work-group. Each work-group is labelled by a unique group ID. A kernel or a memory operation is first enqueued onto a command queue. Kernels are executed asynchronously and the host application execution may proceed right after the enqueue operation. Application may opt to wait for an operation to complete and an operation (kernel or memory) may be marked with a list of events that must occur before it executes. Events are kernel completion and memory operations. OpenCL traverses the dependence graph between the kernels and memory transfers in a queue and ensures the correct execution order. Multiple command queues may be constructed further enhancing parallelism control across problems and multiple devices.

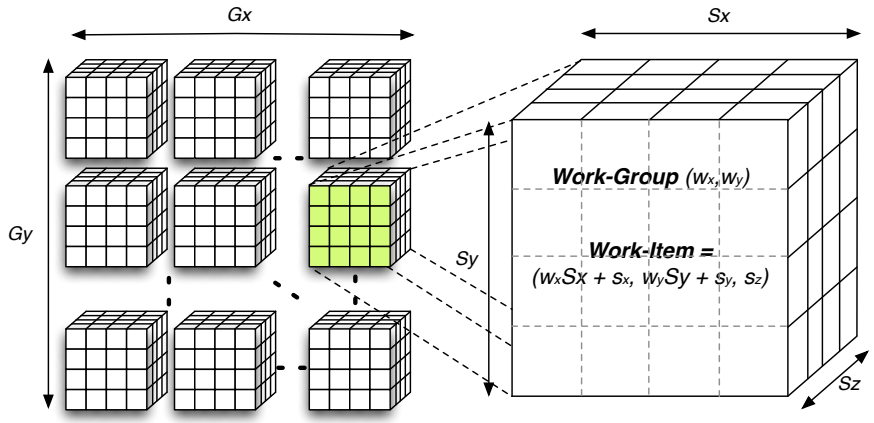


Figure A.7: OpenCL 3D Kernel of size Gx Gy, comprising of work-groups of Sx Sy Sz work-items. *Adapted from OpenCL Specification* [106].

A.4.2.3 *OpenCL Memory Model*

Each compute device has global memory space which typically resides in RAM (either system RAM or on the graphics board) (*cf.* Table A.1). A constant memory is also global but read-only. Constant memory residence is implementation dependent. Each compute unit has local memory which usually resides very near the chip. Local memory is usually much faster and smaller than global memory. This

scarce, fast and read-write resource is useful for communication of work-items inside their work-groups. Each processing element has a private memory, which is shielded from other items' access. Finally, during run-time each processing element is assigned a set of registers, residing directly on the chip. Data that cannot be held in registers is spilled into private memory, which can be very costly. The OpenCL concurrent-readconcurrent-write (CRCW) memory model has so-called relaxed consistency which means that different work-items may see a different view of global memory as the computation proceeds. Within individual work-items reads and writes to all memory spaces are ordered. Synchronization between work-items in a work-group is necessary to ensure consistency. No mechanism for synchronization between work-groups is provided. Such model ensures parallel scalability by requiring explicit synchronisation and communication, thus forcing programmers to write scalable code. The OpenCL specification does not explicitly state where each memory space will be mapped to on individual implementations. This provides great freedom for vendors on one hand and some uncertainty for programmers on the other. Fortunately, kernels may be compiled just-in-time and possible differences may be tackled in run-time.

Table A.1: Memory and Access Policy

	Global	**Constant**	**Local**	**Private**
Host	Dynamic allocation	Dynamic allocation	Dynamic allocation	No allocation
	Read/Write access	Read/Write access	No access	No access
Kernel	No allocation	Static allocation	Static allocation	Static allocation
	Read/Write access	Read-Only access	Read/Write access	Read/Write access

A.4.2.4 *Programming Model: OpenCL Parallelism and Synchronisation*

A barrier mechanism is present for intra-work-group synchronization of work-items inside a kernel and for event synchronization on the host API level. A barrier instruction is an execution point, which must be encountered by all items. OpenCL defines barriers of two types. Command queue barriers are global synchronization points that are enqueued onto command queues and define the events (kernels, memory operations or other barriers) which must all encounter them. Other means of in-kernel synchronization include atomic instructions on local memory or global memory, optionally present in some OpenCL implementations and mandatory since OpenCL 1.1. Inter-work-group synchronization mechanisms in kernels are not available. Task paral-

lelism and synchronization between individual kernels and memory transfer operations are handled by the command queues in API. Barriers are kernel instructions and all work-items which reach the barrier are stalled until the arrival of the last work-item. Kernel barriers are of two main types. The instruction barrier requires all work-items to execute before they are allowed to continue. All work-items of a work-group must encounter this instruction. Instructions mem_fence , read_mem_fence and write_mem_fence order loads and/or stores, which means that pending memory operations will be committed to memory before the execution proceeds.

A.4.2.5 *OpenCL C Language*

OpenCL kernels are programmed using a ISO C99based language. This language misses some C99 features namely standard C99 headers, function pointers, recursion, variable length arrays and bit fields. On the other hand, it supports some new necessary additions including vector data types (e.g. float4 , int2) and address space qualifiers (__constant , __local , __global and the implicit __private) and several other keywords (e.g. __kernel). The OpenCL C Language also includes built-in support for synchronization, work-item and work-group indexing, whole range of mathematical functions and image manipulation routines. Many of the functions are similar to their GLSL counterparts. An example of an OpenCL kernel is shown in Listing 2.2.

ATOMIC INSTRUCTIONS OpenCL 1.0 optional extensions include atomic instructions. Basic atomic instructions include operations such as addition, exchange or decrementation and extended atomic instructions include more complex operations including a minimum, maximum or xor. Atomic instructions2.4. OPENCL 9 operation may be supported on global memory, local memory or both. The transactions are guaranteed to be atomic only for the device executing them, not across devices on OpenCL 1.0.

A.4.2.6 *OpenCL Versions*

The current version of OpenCL is 1.2. Some of the major enhancements and new features compared to the first version are: Host-thread safety. Now it's safe to execute OpenCL API calls from any host thread. Sub-buffer objects distribute buffer regions across multiple devices. Three-component vector data types as in GLSL. Global work-offsets enabling kernels to operate on different portions of the NDRange. Memory destructor callback Memory operations on rectangular subregions of buffers as in GLSL. Several new kernel instructions. Improved OpenGL interoperability. Optional features from OpenCL 1.0

(e.g. atomic instructions, double and half precision support) are new part of the 1.1 core.

B

MODEL-DRIVEN ENGINEERING

According to Stuart Kent [90], Model-Driven Engineering (MDE) [133] is wider in scope than the Object Management Group (OMG)'s Model-Driven Architecture (MDA). The MDA strategy envisages a world where models play a more direct role in software production, being amenable to manipulation and transformation by machine. MDE combines *process* and *analysis* with architecture.

As development in MDE and associated tools and technologies are increasing, it has become a promising approach not only for software engineering but for hardware as well as system engineering, attracting much attention in industry and academia. Furthermore, MDE plays a very important role, which contributes to modeling, automatic code generation and bridging between different technologies.

The key integral concept in MDE is a model. Two core concepts regard a model in MDE. First, the *representation* (a model is a representation of a system); second, *conformance* (a model conforms to a metamodel) [17]. These two concepts are distinctly presented in Section B.1. Another key notion of MDE, model transformation is also discussed in Section B.3.

B.1 MODELS AND METAMODELS

Development of complex software systems using modeling languages to specify models at high-levels of abstraction is the philosophy underlying MDE. In this domain, there are two schools of thought that advocate the development of such modeling languages : general-purpose modeling and domain-specific modeling. As an example, Unified Modeling Language (UML) (*cf.* next Section) is an example of general-purpose modeling with a large number of classes and properties to model various aspects of a software system using the same language. For domain-specific modeling, researchers have been creating Domain-Specific Language (DSL) based metamodels in order to attain this goal. Moreover, extending UML with profiles is another way to aim domain-specific modeling. UML and their profiles are subject to next section. In this section, we present the concept of model and metamodel.

In summary, Figure B.1 presents the relationship among *system*, *model*, and *metamodel*. A particular view (or aspect) of a system can be captured by a model and that each model is written in the language of its metamodel. Systems are represented by models at an abstraction level. A metamodel is also a model. Indeed, a metamodel is a model of a modeling language. A model conforms to a metamodel in order to represent a system. Yet, a metamodel is presented as a diagram that defines the concepts of the language used to model a system. A 4-layer architecture is used to define modeling languages as seen in Table B.1. This provides the the roles general relationship among instances, models, metamodels, and meta-metamodels. These concepts are recurrently referenced in this thesis. They are part of the identification of elements in the process of abstraction and refinement of models.

Table B.1: Modeling Languages Architecture according to MDA

Level	**Description**	**MDA Termin.**	**Example**
3	Language for defining languages	meta-metamodel	A MOF Class (MetaClass)
2	Language definition	metamodel	a UML Class
1	Domain concepts / Language elements	model	Class "Task"
0	Domain/Language instances	instance	Object task = "Multiplication"

B.1.1 *Abstraction and Refinement of Models*

Abstraction is a process that selectively removes some information from a description to focus on the information that remains.

B.2 UML AND PROFILES

The Unified Modeling Language (UML) [115] defines a notation and a metamodel. It is a language for visualizing, specifying, constructing and documenting the artifacts of software systems. It is also a general-purpose modeling language that can be used with all major object methods and applied to all application domains.

Since UML is not a methodology, it does not require any formal work products. Yet it does provide several types of diagrams that, when used within a given methodology, increase the ease of understanding an application under development. There is more to UML than these diagrams. However, we confine the UML concepts to the diagrams that we have used in our methodology as follow:

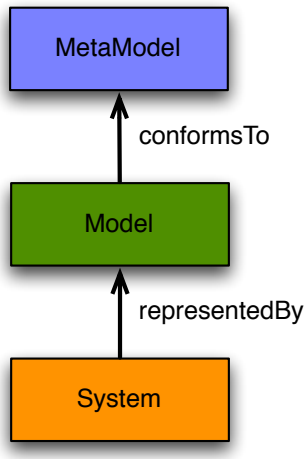


Figure B.1: System, Model, and Metamodel Relationships

- **The Composite Structure Diagram** is one of the artifacts added to UML 2.0. It shows the internal structure (including parts and connectors) of a structured classifier or collaboration. This diagram visualizes the internal structure of a class or collaboration. Composite structure diagram is a kind of component diagram mainly used in modeling a system at micro point-of-view.
- **The Deployment Diagram** helps to model the physical aspect of an Object-Oriented software system. It models the run-time configuration in a static view and visualizes the distribution of components in an application. In most cases, it involves modeling the hardware configurations together with the software components that lived on.

Moreover, below there is the description of some important elements defined in UML. These elements are used in our modeling approach.

- A **class** describes a set of objects that share the same specifications of features, constraints, and semantics. Class is a kind of classifier whose features are attributes and operations. Attributes of a class are represented by instances of Property that are owned by the class. Some of these attributes may represent the navigable ends of binary associations.
- A **part** represents a set of instances that are owned by a containing classifier instance. When an instance of the containing classifier is created, a set of instances corresponding to its parts may be created either immediately or at some later time. These instances are instances of the classifier typing the part. A part specifies that a set of instances may exist; this set of instances is a subset of the total set of instances specified by the classifier typing the part.
- A **port** is a property of a classifier that specifies a distinct interaction point between that classifier and its environment or between

the (behavior of the) classifier and its internal parts. Ports are connected to properties of the classifier by connectors through which requests can be made to invoke the behavioral features of a classifier.

- A **connector** specifies a link that enables communication between two or more instances. Usually they are used to connect two ports.
- A **dependency** is a relationship that signifies that a single or a set of model elements requires other model elements for their specification or implementation. This means that the complete semantics of the depending elements is either semantically or structurally dependent on the definition of the supplier element(s). As an extension, an **abstraction** is a relationship that relates two elements or sets of elements that represent the same concept at different levels of abstraction or from different viewpoints. In the metamodel, an Abstraction is a Dependency in which there is a mapping between the supplier and the client.
- An **artifact** is the specification of a physical piece of information that is used or produced by a software development process, or by deployment and operation of a system. Examples of artifacts include model files, source files, scripts, and binary executable files, a table in a database system, a development deliverable, or a word-processing document, a mail message.
- A **manifestation** is the concrete physical rendering of one or more model elements by an artifact.
- An **instance** specification is extended with the capability of being a deployment target in a deployment relationship, in the case that it is an instance of a node. It is also extended with the capability of being a deployed artifact, if it is an instance of an artifact.

B.2.0.1 *UML Profiles*

In some occasions UML may not be precise enough for modeling specific problem domain. The UML Profile is an extension mechanism for customizing the models for specific domains or platforms. Extension mechanisms allow refining standard semantics in strictly additive manner, so that they cannot contradict standard semantics [4].

Indeed, the profiles package included in UML 2.0 defines a set of UML artifacts that allows the specification of an MetaObject Facility (MOF) model to deal with the specific concepts and notation required in particular application domains (e. g., real-time, business process modeling, finance, etc.) or implementation technologies (e. g., .NET, J2EE, or CORBA). A UML profile is defined as a UML package stereotyped ÒprofileÓ, that can extend either a metamodel or another profile. UML Profiles are defined in terms of three basic mechanisms: *stereotypes, constraints,* and *tagged values.*

- **Stereotype** is a profile class which defines how an existing metaclass may be extended as part of a profile. It enables the use of a platform or domain specific terminology or notation in place of, or in addition to, those used for the extended metaclass.
- **Tagged value** are the values of the properties of a stereotype applied to a model element.
- **Constraint** is a packageable element representing condition, restriction or assertion related to some of the semantics of an element (that owns the constraint).

B.2.1 *Introduction to MARTE*

The UML profile for MARTE (or MARTE profile) [116] extends the possibilities for modeling of application and architecture and their relations. In addition, MARTE allows to extend the performance analysis and task scheduling based on target platform architecture. MARTE consists in defining foundations for model-based description of real time and embedded systems. These core concepts are then refined for both modeling and analyzing concerns. Modeling parts pro- vide support required from specification to detailed design of real-time and embedded characteristics of systems. MARTE concerns also model-based analysis. In this perspective, the intent is not to define new techniques for analyzing real- time and embedded systems, but to support them. Hence, it provides facilities to annotate models with information required to perform specific analysis. Especially, MARTE focuses on performance and schedulability analysis. However, it defines also a general analysis framework which intends to refine/specialize any other kind of analysis. Among others, the benefits of using this profile are thus:

- providing a common way of modeling both hardware and software aspects of a RTES in order to improve communication between developers;
- enabling interoperability between development tools used for specification, design, verification, code generation, etc.;
- fostering the construction of models that may be used to make quantitative predictions regarding real-time and embedded features of systems taking into account both hardware and software characteristics.

Allocation Modeling (Alloc) from Foundations, Generic Resource Modeling (GRM) and Generic Component Model (GCM) from Design Model, and Repetitive Structure Modeling (RSM) annex are packages that provide the main resources to model and to describe our entire application. In particular, RSM provides concepts to allow to express the inherent parallelism of applications.

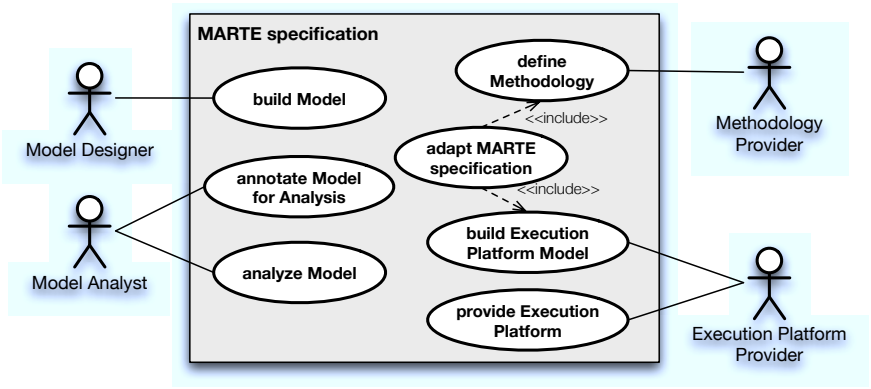


Figure B.2: MARTE Use-Case

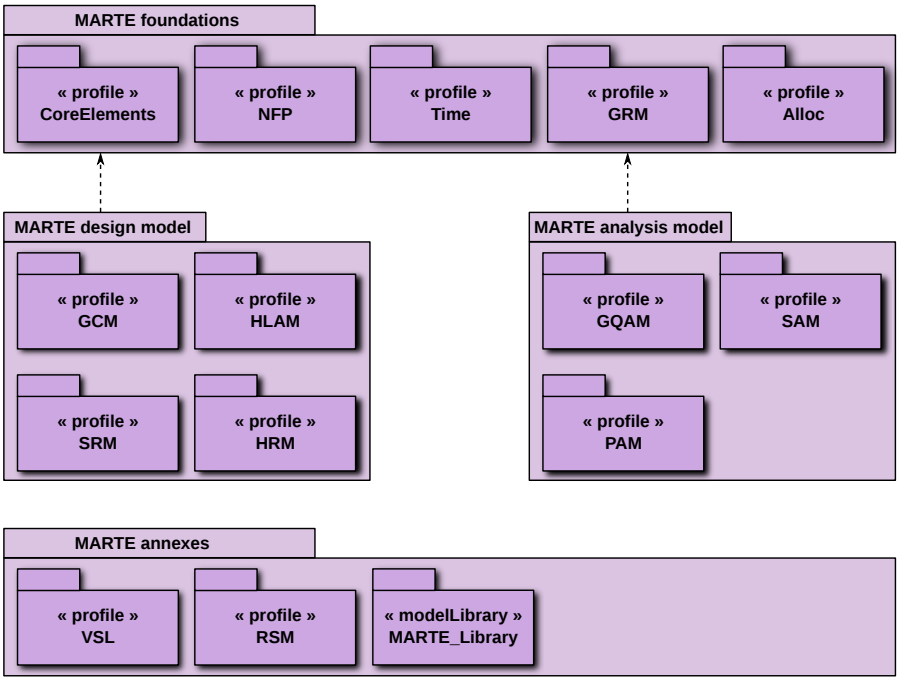


Figure B.3: Architecture of the UML profile for MARTE

B.2.2 *RSM Package and Array Oriented Language (ARRAYOL)*

The RSM package of Modeling and Analysis of Real-Time and Embedded Systems (MARTE) allows to describe regular parallelism in a system: functional algorithms, hardware architectures topologies, allocation of functionality on execution platforms. The main advantage of the RSM concepts is that they provide a factorized and compact representation of potentially large regular structures. This is very helpful for the design scalability, typically when we work with massively parallel systems. An overview of the RSM package is shown in Figure B.4. All its basic stereotypes inherit from the LinkTopology stereotype, which generalizes the nature of the different kinds of links in a regular structure. These links are *Tiler*, *InterRepetition*, *DefaultLink* and *Reshape*. RSM is strongly inspired in ARRAYOL specification language. Since ARRAYOL is an important part of our code generation approach, the remaining of this subsection is dedicated to clarify the most key points of its concepts.

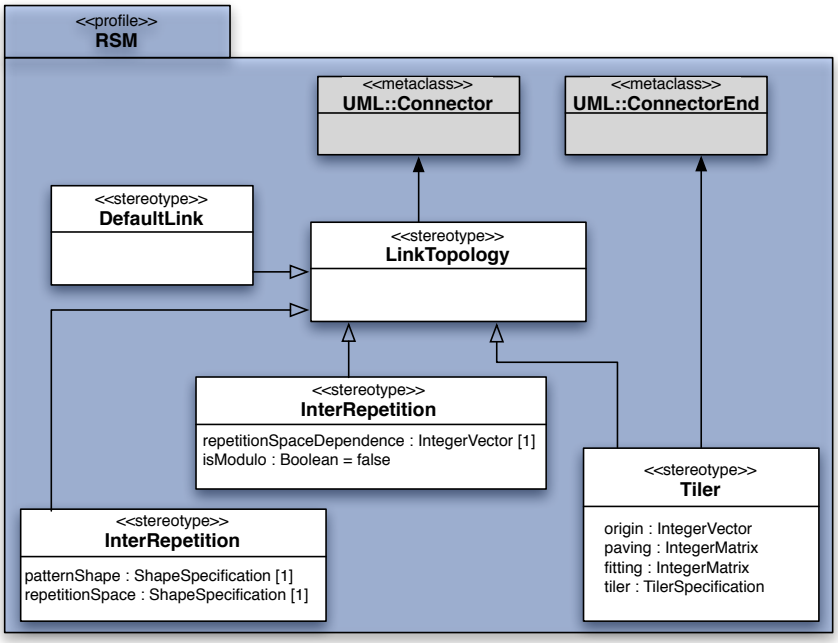


Figure B.4: RSM Package Overview

B.2.2.1 *Specification Language* ARRAYOL

ARRAYOL [45, 18, 19] is a specification language developed by Alain Demeure [3] at Thales Underwater System. This language allows us specifying data-intensive signal processing applications taking into account large data and the potential parallelism of tasks.

Remark: ARRAYOL *is only a specification language, no rules are provided for executing an application described with* ARRAYOL*, but a scheduling policy can be easily computed using this description.*

Initially, this language's designers found that most of the complexities of the specification of parallel applications was the way how applications access their data. For this reason ArrayOL aims only the data dependence specification (either in data or task parallelism), without taking into account the task functionalities. The intended applications usually work on multidimensional arrays. The complexity of these applications does not come from the elementary functions they combine, but from their combination by the way they access the intermediate arrays. Indeed, most of the elementary functions are sums, dot products, solvers, or Fourier transforms, which are well known and often available in functions libraries. The difficulty and the variety of those data-intensive signal processing applications come from the way these elementary functions access their input and output data as parts of multidimensional arrays. The complex access patterns lead to difficulties to schedule these applications efficiently on parallel and distributed execution platforms. As these applications handle large amounts of data under tight real-time constraints, the efficient use of the potential parallelism of the application on parallel hardware is mandatory.

ArrayOL has a single assignment formalism. No data element is ever written twice, even it can be read several times. ArrayOL can be considered as a first order functional language.

B.2.2.2 *Task Parallelism Specification*

The task parallelism is represented by a compound task. The compound description is a simple Directed Acyclic Graph (DAG). Each node represents a task and each edge a dependence connecting two conform ports (same type and shape). There is no relation between the shapes of the inputs and the outputs of a task. So a task can read two two-dimensional arrays and write a three-dimensional one. The creation of dimensions by a task is very useful, a very simple example is the Fast Fourier Transform (FFT) algorithm which creates a frequency dimension.

For instance, we will describe the ArrayOL usage in the *Downscaler* case study seen in Chapter 3 and summarily shown in Figure B.5. The tasks are represented by named rectangles, their ports are squares on the border of the tasks. The shape of the ports is written as a t-uple of positive numbers or ∞. The dependences are represented by connectors between ports and the direction is indicated by following tasks from left to right. Sometimes we can have an additional dimension representing the *time*. For instance, in Figure B.5, if we change $\{288, 352\}$ to $\{288, 352, \infty\}$ we add the infinite (or at least undefined) flow of frames[32] .

With ArrayOL it is not possible to express a *data flow graph*. Indeed, the internal behavior of tasks is not explicit. Thus, this bounds the scheduling definition to data dependence specifications.

[32] *For this application, a frame has the CIF format: 352 rows and 288 lines.*

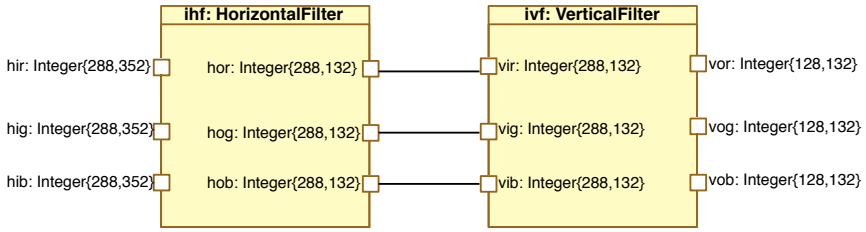


Figure B.5: The task parallelism in ArrayOL. Two tasks

B.2.2.3 *Data Parallelism Specification*

Currently, tasks have the repetition specification from their shapes, ant this also provide information about the data-parallelism. Indeed, each repetition of repeated tasks is reckoned as an independent elementary task[33] . Thus, their overall order (sequential or parallel) does not matter.

[33] *However, there are some cases where inter-repetition dependence can exist.*

Moreover, each instance of the repeated task operates with sub-arrays of the inputs and outputs of the repetition. For a given input or output, all the sub-array instances have the same shape, and they are composed of regularly spaced elements and are also regularly placed in the target array. This hypothesis allows a compact representation of the repetition and is coherent with the application domain of ArrayOL which describes very regular algorithms.

As those sub-arrays have a regular shape, they are called *patterns* when considered as the input arrays of a repeated task and *tiles* when considered as a set of elements of the arrays of the repeated task. In order to give all the necessary information to create these *patterns*, a *tiler* is associated to each array (i. e. each edge). Actually, a *tiler* is able to build the *patterns* from an input array, or to store the *patterns* in an output array. It describes the coordinates of the elements of the *tiles* from the coordinates of the elements of the *patterns*. It contains the following information:

- **o**: *origin*, the global coordinates within the input array used as reference for each gathered pattern;
- **F**: *Fitting*, is the matrix associated to each pattern in order to obtain the elements that compose a pattern;
- **P**: *Paving*, is the matrix associated to each instance of the repetition space of a repeated task in order to point out the origin for its corresponding pattern.

FINDING A REFERENCE FOR A GIVEN PATTERN. For this operation, we use the *origin* and *paving* definitions. In summary, this process is defined in the following formula:

$$\forall \mathbf{r} | (zero_{rep} \leqslant \mathbf{r} < s_{rep}), \mathbf{ref}_r = o + P.\mathbf{r} \mod s_{array} \quad \text{(B.1)}$$

where **r** is the index in the repetition space (comprised between $zero_{rep}$ and s_{rep}), and s_{array} is the multidimensional size of the array. Figure B.6 shows an example of application of the formula (B.1). For each r in the repetition space $s_{rep} = (4)$, we have a new origin point in the array. From this point we can compute the elements of the pattern.

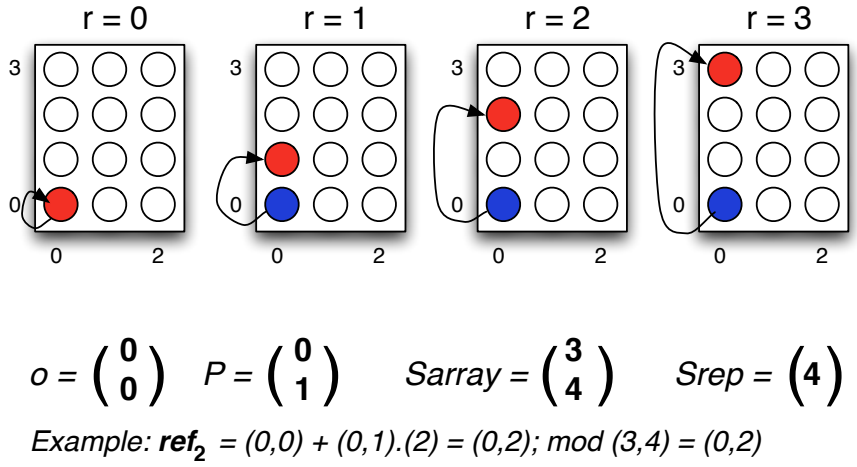


Figure B.6: Paving Example in ArrayOL

DEFINING THE ELEMENTS FOR A GIVEN PATTERN. Once we have the ref defined for each instance of the repetition space, we can point out the elements that will fill[34] the corresponding pattern. For this operation, we use the ref previously obtained and *fitting* definitions. Formula (B.2) and Figure B.7 present the general application and an example based on this definition.

[34] *Here, it depends on the type of tiler. For instance, for input tilers, the elements in the input array are "copied" to the pattern. However, for output tilers elements in the pattern are "copied" to the output array.*

$$\forall \mathbf{i} | (zero_{pat} \leqslant \mathbf{i} < s_{pat}), \mathbf{e}_i = \mathbf{ref} + F.\mathbf{i} \mod s_{array} \tag{B.2}$$

where **i** is the index in the pattern (comprised between $zero_{pat}$ and s_{pat}), and again s_{array} is the multidimensional size of the array.

Both previously defined formulas (B.1)(B.2) can be merged into only a single one giving the index for any element in the pattern knowing its **r** and its **i**, as seen below:

$$\begin{aligned} &\forall (\mathbf{r},\mathbf{i}) | ((zero_{rep} \leqslant \mathbf{r} < s_{rep}), (zero_{pat} \leqslant \mathbf{i} < s_{pat})), \\ &\mathbf{e}_{r,i} = o + \begin{pmatrix} P & F \end{pmatrix} . \begin{pmatrix} \mathbf{r} \\ \mathbf{i} \end{pmatrix} \mod s_{array} \end{aligned} \tag{B.3}$$

B.2.2.4 *ArrayOL Execution Model*

Since ArrayOL is a specification language, there is no execution model specifically defined for it. Philippe Dumont, in his thesis [49],

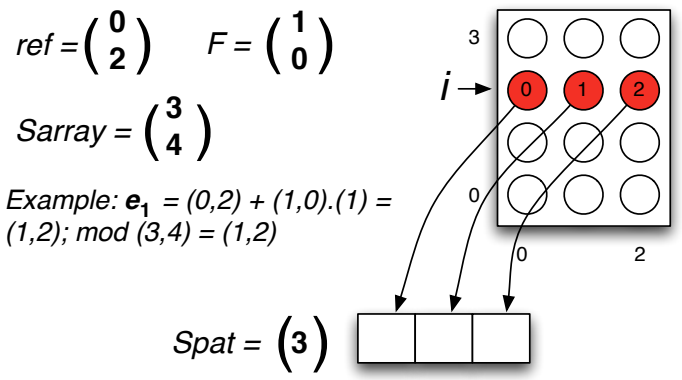


Figure B.7: Pattern Distribution in ARRAYOL

wrote a study about possible execution models supported by ARRAYOL. Here, we emphasizes three of those models:

- sequential: all tasks in a task graph as well as repetitions of tasks execute sequentially regarding their data dependence;
- SPMD: all tasks ready to execute can do it taking advantage of parallel systems;
- pipeline: the whole system can execute in parallel according to a data flow, tasks are executed as soon as its input data is available.

As seen in Chapter A, GPUs are SPMD rather than SIMD since a complete program is run on multiple values. Thus, ARRAYOL is well suitable to our execution model.

B.3 MODEL TRANSFORMATION

Model Transformations [81, 70] are a key prerequisite for MDE and therefore represent an active research area. Various model transformation languages are available, whereas the languages can be categorized into different approaches. The basic idea and coherence of model transformation can be seen in Figure B.8. The most basic entity is the meta-metamodel, as can be seen at the top of the illustration. This is called 3rd level of the hierarchy. The meta-metamodel only contains basic elements for a universal language, which is suffiient for specifying metamodels (*cf.* Table B.1). In MDA, which is the model driven engineering approach by OMG, MetaObject Facility (MOF) is used as meta-metamodel.

Besides metamodels that describe input and output models, a special metamodel resides at level 2, which is specific for model transformation, namely MMT. MMT denotes the metamodel of the model transformation language used. Although most transformation languages use textual syntax rather than MOF-based models, many model transformation languages still are specified using a meta-metamodel

like MOF, like it is the case with Query/View/Transformation (QVT) [70, 73] and Atlas Transformation Language (ATL) [85]. In such cases, the textual representation is interpreted as concrete syntax. At level 1, the models themselves, in this case Model1 and Model2, can be found. Models at the 1st level are able to use constructs defined in the metamodels at level M2. From a model transformation point of view, a specific transformation also resides at this level, because it is an instance of a model transformation language metamodel, and therefore uses the features provided by the metamodel. A model transformation uses models at the same level as input and output models. For example, a model transformation written in QVT is conform to the QVT metamodel at level 2 and operates on models at level 1, which themselves again are conform to other metamodels at level 2. Thus, a Model-to-Model Transformation (M2M) uses often a model as source and/or target and sometimes the same model as inout model (source and target at the same time).

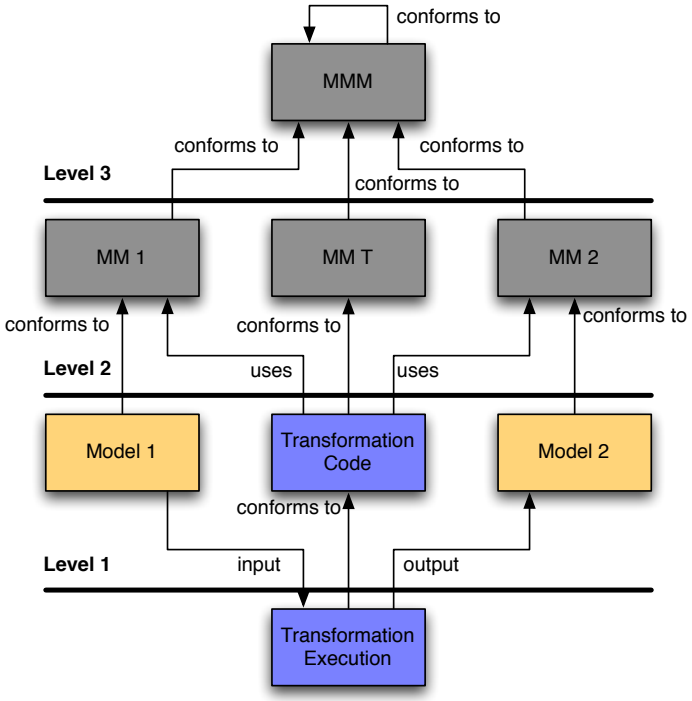


Figure B.8: Model Transformation Pattern.*Based on [81]*

In our works we do not use ATL. We have chosen QVT aiming at retaining compatibility with previous modules and easy handling. Furthermore, ATL has many similarities with QVT. QVT is actually a collection of three transformation languages, but only the Operational Mapping Language (OML) was used in our implementations.

B.3.1 *Model Refactoring*

Model transformation can be used for model refactoring, specifically through the use of in-place transformations. Using a transformation defined on a model that targets the same instance as the input model, it is possible to create complex refactorings. Model refactoring is again mentioned in Chapter 2 as part of the refactoring and model transformations strategies chosen by Gaspard2.

B.3.2 *Model Merge*

QVT Operational Mapping Language (introduced in next subsection) is capable of dealing with multiple input and output models. One application of this is for merging models taking elements from several input models and creating another one containing elements transformed from different sources. This is an interesting aspect for our approach because this allows us creating a model library of static elements.

B.3.3 *M2M QVT Operational Mapping Language*

As an Object Constraint Language (OCL) [117]-based procedural language, OML provides a full method for defining M2M transformations. Actually, OCL is used in the "Query" from QVT. Much as with a Java class file, an OML definition (*.qvto file) usually comprehends a list of imported models, a main operation, and a series of mappings and queries (like class methods). Aside from the need to be familiar with the OCL and some extensions added to produce side effects, OML should be fairly easy for most developers to get started using. Those already familiar with OCL should find it much easier to use.

B.3.3.1 *Mapping operations as refinements of relations*

A transformation specifies a set of relations that must hold between the model elements of a set of candidate models. These models are named, and the types of elements they can contain are restricted to those within a set of referenced packages. A transformation execution is in a particular direction, which is defined as the selection of one of the models as a target. The execution of this kind of relational transformation proceeds by attempting to make all the relations hold by modifying only the target model.

An *Operational Transformation* is defined as a refinement of a relational transformation on a given direction. It is invokable explicitly using a signature which defines, firstly, the constraints for a model to be a valid participant and, secondly, the role played by each model in the transformation (*input/output/inout* classification). An *Operational*

Transformation, in turn, defines *MappingOperations* which are defined as refinements of *Relations* for a given direction. In addition it defines a *main* operation which provides an entry point for the execution of the transformation.

A *Mapping Operation* is syntactically described by a signature, a guard (a *when* clause), a mapping body and a postcondition (a *where* clause). Conceptually, a mapping operation refines a relation which is the owner of the *when* and *where* clauses. In addition, the relation imposes constraints on the signature of the operation, which should be consistent with the relation domain types.

A mapping may have three different sections: an initialization section, a population section and a termination section. None of three sections is mandatory. Also a mapping may declare a guard (when clause) to restrict the applicability of the rule and a postcondition (where clause). The general notation, when there is no shorthand applying, is seen in Listing B.1:

Listing B.1: Launchtopology Computation directly from Task's Shape

```
1 mapping dirkind0 X::mymapping (dirkind1 p1:P1, dirkind2 p2:P2) : r1:
      R1, r2:R2
2   when { ... }
3   where { ... }
4 {
5   init { ... }
6   population { ... }
7   end { ... }
8 }
```

Where *dirkind* refers to *in/inout/out* direction kinds.

B.4 CODE GENERATION

In order to generate code, we have to transform the world of models to text. Model-to-Text Transformation (M2T) enables transformations of models to various text artifacts such as code, deployment specifications, reports and application documentation.

While QVT standard addresses the needs of model-to-model transformation, the MOF Model to Text (MOFM2T) [114] standard addresses how to translate a model to various text artifacts such as code, deployment specifications, reports, documents, etc. Essentially, the MOFM2T standard needs to address how to transform a model into a linearized text representation. An intuitive way to address this requirement is a template based approach wherein the text to be generated from models is specified as a set of text templates that are parameterized with model elements. A template-based approach is used wherein a *template* specifies a text template with placeholders for data to be extracted from models. These placeholders are essentially expressions specified over metamodel entities with queries being the primary mechanisms

for selecting and extracting the values from models. These values are then converted into text fragments using an expression language augmented with a string manipulation library. Templates can be composed to address complex transformation requirements. Large transformations can be structured into modules having public and private parts.

In 2006, the project Acceleo is born under the GNU Public Licence (GPL). In 2009, while moving to Acceleo 3, the project has been accepted in the Eclipse Foundation. During this transition, the language used by Acceleo to define a code generator has been changed to use the standard from the OMG for model to text transformation, MOFM2T. In the next chapter, we present again Acceleo, however in the Gaspard2 context.

BIBLIOGRAPHY

[1] Adolf Abdallah. *Conception de SoC à Base d'Horloges Abstraites : Vers l'Exploration d'Architectures en MARTE*. PhD thesis, Université des Sciences et Technologie de Lille - Lille I, March 2011. in French. (Cited on page 39.)

[2] AccelerEyes. Jacket on Matlab. http://www.accelereyes.com/products/jacket, 2011. (Cited on page 27.)

[3] Alain Demeure and Anne Lafage and Emmanuel Boutillon and Didier Rozzonelli and Jean-Claude Dufourd and Jean-Louis Marro. Array-OL : Proposition d'un Formalisme Tableau pour le Traitement de Signal Multi-Dimensionnel. In *Gretsi*, Juan-Les-Pins, France, September 1995. in French. (Cited on page 193.)

[4] S.S. Alhir. *Guide to applying the UML*. Springer professional computing. Springer, 2002. (Cited on page 190.)

[5] Vincent Aranega, Jean-Marie Mottu, Anne Etien, and Jean-Luc Dekeyser. Traceability mechanism for error localization in model transformation. In *proceedings of the ICSOFT 2009 Conference*, Sofia, Bulgaria, July 2009. (Cited on page 108.)

[6] Vincent Aranega, Anne Etien, and Jean-Luc Dekeyser. Using an Alternative Trace for QVT. In *Workshop on Multi-Paradigm Modeling*, Olso, Norway, October 2010. (Cited on page 108.)

[7] Vincent Aranega, Jean-Marie Mottu, Anne Etien, and Jean-Luc Dekeyser. Traceability for Mutation Analysis in Model Transformation. In Juergen Dingel and Arnor Solberg, editors, *Models in Software Engineering*, volume 6627 of *Lecture Notes in Computer Science*, pages 259–273. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-21209-3. (Cited on page 108.)

[8] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2001. ISBN 1558606742. (Cited on page 40.)

[9] Rabie Ben Atitallah. *Modèles et simulation de systèmes sur puce multiprocesseurs - Estimation des performances et de la consommation d'énergie*. PhD thesis, Université des Sciences et Technologie de Lille, December 2007. in French. (Cited on page 39.)

[10] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst.

Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition. SIAM, Philadelphia, PA, 1994. (Cited on page 131.)

[11] J. Bastos and N. Sadowski. *Electromagnetic modeling by finite element methods*. Electrical engineering and electronics. Marcel Dekker, 2003. ISBN 9780824742690. (Cited on page 130.)

[12] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008. (Cited on page 145.)

[13] Rabie Ben Atitallah, Lossan Bonde, Smail Niar, Meftali, and Samy Jean-Luc Dekeyser. Multilevel MPSoC Performance Evaluation Using MDE Approach. In *International Symposium on System-on-Chip 2006 (SOC 2006)*, Tampere, Finland, November 2006. (Cited on page 36.)

[14] Siegfried Benkner. Optimizing irregular hpf applications using halos. *Concurrency - Practice and Experience*, 12(2-3):137–155, 2000. (Cited on page 132.)

[15] Beowulf.org. Beowulf Project. http://www.beowulf.org, 2011. (Cited on page 169.)

[16] Natacha BEREUX. "code_Carmel3D": mise en oeuvre de la version 1.0. Research Report H-R25-2008-03705-FR, EDF R&D, Oct 2008. In French. (Cited on page 131.)

[17] Jean Bézivin. On the unification power of models. *Software and System Modeling*, 4(2):171–188, 2005. (Cited on page 187.)

[18] Pierre Boulet. Array-OL Revisited, Multidimensional Intensive Signal Processing Specification. Research Report RR-6113, INRIA, February 2007. URL http://hal.inria.fr/inria-00128840_v3. (Cited on page 193.)

[19] Pierre Boulet. Formal Semantics of Array-OL, a Domain Specific Language for Intensive Multidimensional Signal Processing. Research Report RR-6467, INRIA, March 2008. URL http://hal.inria.fr/inria-00261178_v2. (Cited on page 193.)

[20] Pierre Boulet. Formal Semantics of Array-OL, a Domain Specific Language for Intensive Multidimensional Signal Processing. Technical report, INRIA, march 2008. Research Report RR-6467 in http://hal.inria.fr/inria-00261178_v2. (Cited on page 36.)

[21] Emmanuel Cagniot. *Algorithmes Data-parallèles Irréguliers Appliqués à la Simulation Éléctromagntique Tridimensionelle*. PhD thesis, Université des Sciences et Technologie de Lille, December 2000. in French. (Cited on page 132.)

[22] CAPS. HMPP. http://www.caps-enterprise.org, 2011. (Cited on page 24.)

[23] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, pages 178–188, New York, NY, USA, 1987. ACM. ISBN 0-89791-215-2. (Cited on page 39.)

[24] CEA. Papyrus - Open Source Tool for Graphical UML2 Modeling. http://www.papyrusuml.org, 2011. (Cited on page 113.)

[25] CEA and EDF and OpenCASCADE. SALOME - The Open Source Integration Platform for Numerical Simulation. http://www.salome-platform.org, 2011. (Cited on pages 133 and 147.)

[26] Ali Cevahir, Akira Nukada, and Satoshi Matsuoka. High performance conjugate gradient solver on multi-gpu clusters using hypergraph partitioning. *Computer Science - Research and Development*, 25:83–91, 2010. ISSN 1865-2034. URL http://dx.doi.org/10.1007/s00450-010-0112-6. 10.1007/s00450-010-0112-6. (Cited on page 158.)

[27] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007. ISBN 0262533022, 9780262533027. (Cited on page 175.)

[28] Indranil Chowdhury and Jean-Yves L'Excellent. Some Experiments and Issues to Exploit Multicore Parallelism in a Distributed-Memory Parallel Sparse Direct Solver. Research Report RR-7411, INRIA, 2010. Research Report RR-7411 in http://hal.inria.fr/inria-00524249. (Cited on page 135.)

[29] Arthur I. Cohen. Rate of Convergence of Several Conjugate Gradient Algorithms. *SIAM Journal on Numerical Analysis*, pages 248–259, June 1972. (Cited on page 147.)

[30] Consortium Scilab (DIGITEO). Scilab. http://www.scilab.org, 2011. (Cited on page 2.)

[31] Inc Cray Research. *CRAY-1 computer system reference manual 224004*. Cray Research, Inc., 1976. URL http://books.google.com/books?id=vNliGwAACAAJ. (Cited on page 171.)

[32] E. de Jong, E.M. Paalvast, H.J. Sips, and M.R. van Steen. High-level specification tools for parallel application development. In *CompEuro '92 . 'Computer Systems and Software Engineering',Proceedings.*, pages 163 –168, may 1992. doi: 10.1109/CMPEUR.1992.218516. (Cited on page 5.)

[33] Jacques Chassin de Kergommeaux and Philippe Codognet. Parallel logic programming systems. *ACM Comput. Surv.*, 26: 295–336, September 1994. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/185403.185453. URL http://doi.acm.org/10.1145/185403.185453. (Cited on page 2.)

[34] César Olavo De Moura Filho, Anne Etien, Julien Taillard, Cedric Dumoulin, and Frédéric Guyomarc'H. Component-based Models Going Generic : the MARTE Case-Study. Research Report RR-6632, INRIA, 2008. (Cited on pages 162 and 164.)

[35] Antonio Wendell De Oliveira Rodrigues, Frédéric Guyomarc'h, Yvonnick Le Menach, and Jean-Luc Dekeyser. Parallel Sparse Matrix Solver on the GPU Applied to Simulation of Electrical Machines. In *Compumag 2009*, Florianopolis, Brazil, November 2009.

[36] Antonio Wendell De Oliveira Rodrigues, Vincent Aranega, Anne Etien, Frédéric Guyomarc'h, and Jean-Luc Dekeyser. Enabling Traceability in an MDE Approach to Improve Performance of GPU Applications. Rapport de recherche RR-7720, INRIA, August 2011. Research Report RR-7720 in http://hal.inria.fr/inria-00617912.

[37] Antonio Wendell De Oliveira Rodrigues, Frédéric Guyomarc'h, and Jean-Luc Dekeyser. Using ArrayOL to Identify Potentially Shareable Data in Thread Work-Groups of GPUs. In *Designing for Embedded Parallel Computing Platforms: Architectures, Design Tools, and Applications on DATE 2011*, Grenoble, France, March 2011. Work in-Progress Poster.

[38] Antonio Wendell De Oliveira Rodrigues, Frédéric Guyomarc'h, and Jean-Luc Dekeyser. Programming Massively Parallel Architectures using MARTE: a Case Study. In *2nd Workshop on Model Based Engineering for Embedded Systems Design (M-BED) on Date Conference 2011*, Grenoble, France, March 2011.

[39] Antonio Wendell De Oliveira Rodrigues, Frédéric Guyomarc'h, and Jean-Luc Dekeyser. An MDE Approach for Automatic Code Generation from MARTE to OpenCL. Research Report RR-7525, INRIA, February 2011. Research Report RR-7525 in http://hal.inria.fr/inria-00563411.

[40] Antonio Wendell De Oliveira Rodrigues, Frédéric Guyomarc'h, and Jean-Luc Dekeyser. A Modeling Approach based on UML/MARTE for GPU Architecture. In *Symposium en Architectures nouvelles de machines (SympA'14)*, Saint Malo, France, May 2011.

[41] Antonio Wendell De Oliveira Rodrigues, Frédéric Guyomarc'h, Jean-Luc Dekeyser, and Yvonnick Le Menach. Automatic Multi-

GPU Code Generation applied to Simulation of Electrical Machines. In *Compumag 2011*, Sydney, Australia, July 2011. (Cited on pages xxiv, 156, and 157.)

[42] Antonio Wendell De Oliveira Rodrigues, Frédéric Guyomarc'h, and Jean-Luc Dekeyser. An MDE Approach for Automatic Code Generation from UML/MARTE to OpenCL. *IEEE Computer in Science & Engineering - Special Edition on GPUs, Journal*, Jan 2012.

[43] Antonio Wendell De Oliveira Rodrigues, Frédéric Guyomarc'h, Yvonnick Le Menach, and Jean-Luc Dekeyser. Automatic Multi-GPU Code Generation applied to Simulation of Electrical Machines. *Magnetics, IEEE Transactions on*, 48(2):831 –834, Feb. 2012. ISSN 0018-9464. doi: 10.1109/TMAG.2011.2179527.

[44] M. M. Dehnavi, D. M. Fernandez, and D. Giannacopoulos. Enhancing the Performance of Conjugate Gradient Solvers on Graphic Processing Units. *IEEE Transactions on Magnetics*, 47: 1162–1165, May 2011. doi: 10.1109/TMAG.2010.2081662. (Cited on page 164.)

[45] A. Demeure and Y Del Gallo. An Array Approach for Signal Processing Design. In *In Sophia-Antipolis conference on MicroElectronics (SAME'98): SoC Session*, Sophia-Antipolis, France, 1998. (Cited on page 193.)

[46] I. Dietrich, R. German, H. Koestler, and U. Ruede. Modeling Multigrid Algorithms for Variational Imaging. In *Software Engineering Conference (ASWEC), 2010 21st Australian*, pages 224 –234, april 2010. doi: 10.1109/ASWEC.2010.16. (Cited on page 22.)

[47] C. V. Dodd and W. E. Deeds. Analytical solutions to eddy-current probe-coil problems. *Journal of Applied Physics*, pages 2829–2832, 1968. (Cited on page 129.)

[48] J.J. Dongarra. *Numerical linear algebra for high-performance computers*. Software, environments, tools. Society for Industrial and Applied Mathematics, 1998. ISBN 9780898714289. (Cited on page 48.)

[49] Philippe Dumont. *Spécification Multidimensionnelle pour le Traitement du Signal Systématique*. PhD thesis, Université des Sciences et Technologie de Lille, 2005. in French. (Cited on page 196.)

[50] John W. Eaton, David Bateman, and Søren Hauberg. *GNU Octave Manual Version 3*. Network Theory Ltd., 3 edition, Aug 2008. ISBN 0-9546120-6-X. (Cited on page 27.)

[51] Eclipse Foundation. Eclipse Projects. http://www.eclipse.org, 2011. (Cited on page 43.)

[52] R.S. Elliott. *Electromagnetics: History, Theory, and Applications*. The IEEE/OUP Series on Electromagnetic Wave Theory (Formerly IEEE Only), Editor Series. John Wiley & Sons, 1999. ISBN 9780780353848. (Cited on page 126.)

[53] Timo Euler. *Consistent Discretization of Maxwell's Equations on Polyhedral Grids*. PhD thesis, TU Darmstadt, November 2007. URL http://tuprints.ulb.tu-darmstadt.de/895/. (Cited on pages xxii and 127.)

[54] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *Computers, IEEE Transactions on*, C-21(9):948 –960, sept. 1972. ISSN 0018-9340. doi: 10.1109/TC.1972.5009071. (Cited on page 170.)

[55] Steve Furber. *ARM System-on-Chip Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000. ISBN 0201675196. (Cited on page 173.)

[56] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004. (Cited on page 2.)

[57] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh. Parallel Algorithms for Dense Linear Algebra Computations. *SIAM Rev.*, 32:54–135, March 1990. ISSN 0036-1445. doi: 10.1137/1032002. URL http://portal.acm.org/citation.cfm?id=78843.78845. (Cited on page 48.)

[58] Abdoulaye Gamatié, Eric Rutten, Huafeng Yu, Pierre Boulet, and Jean-Luc Dekeyser. Synchronous Modeling and Analysis of Data Intensive Applications. *EURASIP Journal on Embedded Systems*, 2008. (Cited on page 36.)

[59] Abdoulaye Gamatié, Sébastien Le Beux, Éric Piel, Rabie Ben Atitallah, Anne Etien, Philippe Marquet, and Jean-Luc Dekeyser. A Model Driven Design Framework for Massively Parallel Embedded Systems. *ACM Transactions on Embedded Computing Systems*, 2011. To appear. See also the INRIA research report entitled "A Model Driven Design Framework for High Performance Embedded Systems" http://hal.inria.fr/inria-00311115. (Cited on page 34.)

[60] S. Georgescu and H. Okuda. Conjugate Gradients on Graphic Hardware: Performance and Feasibility. *Lecture Notes in Computer Science*, 2011. is under review and can be found in http://para08.idi.ntnu.no/docs/submission_74.pdf. (Cited on page 158.)

[61] Anne Geraci. *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*. IEEE Press, Piscataway, NJ, USA, 1991. ISBN 1559370793. (Cited on page 42.)

[62] Calin Glitia, Pierre Boulet, Eric Lenormand, and Michel Barreteau. Repetitive Model Refactoring Strategy for the Design Space Exploration of Intensive Signal Processing Applications. *Journal of Systems Architecture*, January 2011. doi: 10.1016/j.sysarc.2010.12.002. (Cited on page 40.)

[63] Flori Glitia, Anne Etien, and Cedric Dumoulin. Fine Grained Traceability for an MDE Approach of Embedded System Conception. In *ECMDA Traceability Workshop*, pages 27–38, Berlin, Germany, 2008. (Cited on page 42.)

[64] Gene H. Golub and Charles F. Van Loan. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996. ISBN 0-8018-5414-8. (Cited on page 138.)

[65] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005. ISBN 0321246780. (Cited on page 26.)

[66] GP-you Group. GPUmat User Guide. http://gp-you.org/download/PDF/GPUmat_User_Guide_0.27.pdf, 2011. (Cited on page 27.)

[67] Daniel Cabeza Gras and Manuel V. Hermenegildo. Non-strict independence-based program parallelization using sharing and freeness information. *Theor. Comput. Sci.*, 410:4704–4723, November 2009. ISSN 0304-3975. doi: 10.1016/j.tcs.2009.07.044. URL http://dl.acm.org/citation.cfm?id=1628316.1628387. (Cited on page 2.)

[68] Anne Greenbaum. Estimating the attainable accuracy of recursively computed residual methods. *SIAM J. Matrix Anal. Appl.*, 18:535–551, July 1997. ISSN 0895-4798. doi: 10.1137/S0895479895284944. URL http://dl.acm.org/citation.cfm?id=263197.263199. (Cited on page 151.)

[69] J.P. Gregoire, C. Rose, and B. Thomas. Direct and iterative solvers for finite-element problems. *Numerical Algorithms*, 16:39–53, 1997. ISSN 1017-1398. URL http://dx.doi.org/10.1023/A:1019174710859. 10.1023/A:1019174710859. (Cited on page 131.)

[70] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 1 edition, 2009. ISBN 0321534077, 9780321534071. (Cited on pages 197 and 198.)

[71] William Gropp, Ewing Lusk, and Thomas Sterling, editors. *Beowulf Cluster Computing with Linux*. MIT Press, 2nd edition, 2003. (Cited on page 169.)

[72] Thorsten Grotker. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002. ISBN 1402070721. (Cited on page 39.)

[73] Object Management Group. Meta object facility (mof) 2.0 query/view/transformation. Specification Version 1.0, Object Management Group, April 2008. (Cited on pages 44 and 198.)

[74] Paul Le Guernic. Signal: A formal design environment for real-time systems. In *TAPSOFT*, pages 789–790, 1995. (Cited on page 39.)

[75] Jing Guo, Antonio Wendell De Oliveira Rodrigues, Jerarajan Thiyagalingam, Frédéric Guyomarc'h, Pierre Boulet, and Sven-Bodo Scholz. Harnessing the Power of GPUs without Losing Abstractions in SaC and ArrayOL: A Comparative Study. In *HIPS 2011, 16th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Anchorage (Alaska), United States of America, May 2011. (Cited on page 62.)

[76] Martin H. Gutknecht and Zdenvek Strakos. Accuracy of two three-term and three two-term recurrences for krylov space solvers. *SIAM J. Matrix Anal. Appl.*, 22:213–229, April 2000. ISSN 0895-4798. doi: 10.1137/S0895479897331862. URL http://dl.acm.org/citation.cfm?id=587703.587731. (Cited on page 151.)

[77] Scott Hauck and André DeHon, editors. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Systems-on-Silicon. Elsevier, 2008. (Cited on page 40.)

[78] Rainer Heintzmann. CudaMAT. http://wwwuser.gwdg.de/~rheintz/Nanoimaging/CudaMat/CudaMat.html, 2011. (Cited on page 27.)

[79] Hestenes, M. R. and Stiefel, E. Methods of conjugate gradients for solving linear system. *J. Res. Nat. Bur. Standards*, B-49:409–436, 1952. (Cited on page 138.)

[80] Jozef Hooman, Nataliya Mulyar, and Ladislau Posta. Coupling Simulink and UML Models. In *Proceedings of Symposium FORMS/FORMATS 2004, Formal Methods for Automation*

and Safety in Railway and Automotive Systems, pages 304–311, Grenoble, France, 2004. (Cited on page 20.)

[81] Philipp Huber. The Model Transformation Language Jungle - An Evaluation and Extension of Existing Approaches. Master's thesis, Business Informatics Group - Institut für Softwaretechnik und Interaktive Systeme, Vienna, AT, 2003. (Cited on pages 197 and 198.)

[82] N. Ida and J. Bastos. *Electromagnetics and calculation of fields*. Springer, 1997. ISBN 9780387948775. (Cited on page 130.)

[83] IEEE. IEEE Std. 1003.1c-1995 thread extensions. Standard, Institute of Electrical and Electronics Engineers, 1996. (Cited on page 38.)

[84] JOCL. Java bindings for OpenCL. http://www.jocl.org, 2011. (Cited on page 26.)

[85] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Sci. Comput. Program.*, 72: 31–39, June 2008. ISSN 0167-6423. (Cited on page 198.)

[86] JScience. Java tools and libraries for the advancement of sciences. http://www.jscience.org, 2011. (Cited on page 1.)

[87] Michio Kaku. *Physics of the Future: How Science Will Shape Human Destiny and Our Daily Lives by the Year 2100*. Knopf Doubleday Publishing Group, March 2011. (Cited on page 176.)

[88] Kantorovich, L. V. and Krylov, V. I. *Approximate Methods of Higher Analysis*. Interscience Publishers, New York, 1958. (Translated from the fourth Russian edition by Curtis D. Bensterl). (Cited on page 138.)

[89] Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of high performance fortran: an historical object lesson. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 7–1–7–22, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-7. doi: http://doi.acm.org/10.1145/1238844.1238851. (Cited on page 132.)

[90] Stuart Kent. Model driven engineering. In *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, pages 286–298, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43703-7. (Cited on page 187.)

[91] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010. ISBN 0123814723, 9780123814722. (Cited on page 178.)

[92] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan C. Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *CoRR*, abs/0911.3456, 2011. (Cited on page 28.)

[93] Yu kwong Kwok, Ishfaq Ahmad, and Ishfaq Ahmad. Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7:506–521, 1996. (Cited on page 87.)

[94] Lawrence Livermore National Laboratory. ROSE Co-anchorler. http://www.rosecompiler.org, 2011. (Cited on page 23.)

[95] Sébastien Le Beux. *Un flot de conception pour applications de traitement du signal systématique implémentées sur FPGA à base d'Ingénierie Dirigée par les Modèles*. PhD thesis, Université des Sciences et Technologie de Lille, December 2007. in French. (Cited on page 40.)

[96] Sébastien Le Beux, Philippe Marquet, and Jean-Luc Dekeyser. A Design Flow to Map Parallel Applications onto FPGAs. In *17th IEEE International Conference on Field Programmable Logic and Applications*, Amsterdam, Netherlands, August 2007. (Cited on page 36.)

[97] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. *SIGPLAN Not.*, 44:101–110, February 2009. ISSN 0362-1340. (Cited on page 175.)

[98] David Lugato, Jean-Michel Bruel, and Ileana Ober. Model-Driven Engineering for High-Performance Computing Applications. In Cakaj Shkelzen, editor, *Modeling Simulation and Optimization - Focus on Applications*, pages 19–33. IN-TECH, http://intechweb.org/, mars 2010. (Cited on page 19.)

[99] MathWorks Inc. Matlab and Simulink. http://www.mathworks.com, 2011. (Cited on pages 2 and 19.)

[100] MathWorks Inc. MATLAB GPU Computing with NVIDIA CUDA-Enabled GPUs. http://www.mathworks.com/discovery/matlab-gpu.html, 2011. (Cited on page 27.)

[101] T. Mattson and M. Wrinn. Parallel programming: Can we PLEASE get it right this time? In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 7 –11, june 2008. (Cited on page 3.)

[102] J.C. Maxwell. *A Treatise on Electricity and Magnetism*. Oxford: Clarendon Press, 1998. (Cited on page 126.)

[103] Karl Meerbergen, Krešimir Fresl, and Toon Knapen. C++ bindings to external software libraries with examples from blas, lapack, umfpack, and mumps. *ACM Trans. Math. Softw.*, 36:22:1–22:23, August 2009. ISSN 0098-3500. doi: http://doi.acm.org/10.1145/1555386.1555391. URL http://doi.acm.org/10.1145/1555386.1555391. (Cited on page 1.)

[104] Microsoft Corp. GPGPU Computing Horizons: Developing and Deploying for Microsoft Windows. Technical Report MSFT_GPGPU_whitepaper_FINAL, Microsoft Research, 2010. (Cited on page 5.)

[105] Gordon E. Moore. Progress in Digital Electronics. In *Technical Digest of the Intel Electronic Devices Meeting*, page 13. IEEE Press, 1975. (Cited on page 176.)

[106] Aaftab Munshi. The OpenCL Specification 1.1. Specification, Khronos Group, January 2011. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf. (Cited on pages 180 and 182.)

[107] G.S. Murthy, M. Ravishankar, M.M. Baskaran, and P. Sadayappan. Optimal loop unrolling for gpgpu programs. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –11, april 2010. doi: 10.1109/IPDPS.2010.5470423. (Cited on page 30.)

[108] San Murugesan. Harnessing green it: Principles and practices. *IT Professional*, 10:24–33, January 2008. ISSN 1520-9202. doi: 10.1109/MITP.2008.10. URL http://dl.acm.org/citation.cfm?id=1344234.1344284. (Cited on page 4.)

[109] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996. ISBN 1-56592-115-1. (Cited on page 38.)

[110] NVIDIA. NVIDIA's Tegra 2 Specification. http://www.nvidia.com/object/tegra-2.html, October 2011. (Cited on page 173.)

[111] NVIDIA Corporation. OpenCL Programming Guide, 2011. Version 4.0. (Cited on pages 49 and 50.)

[112] A. Wendell O. Rodrigues, Frédéric Guyomarc'H, and Jean-Luc Dekeyser. Programming Massively Parallel Architectures using MARTE: a Case Study. In *2nd Workshop on Model Based Engineering for Embedded Systems Design (M-BED 2011) on Date Conference 2011*, Grenoble, France, March 2011. URL http://hal.inria.fr/inria-00578646/en/. (Cited on page 113.)

[113] Obeo. Acceleo Code Generator. http://www.acceleo.org, 2011. (Cited on page 44.)

[114] Object Management Group. MOF Model To Text Transformation Language (MOFM2T). http://www.omg.org/spec/MOFM2T/1.0, 2008. Version 1.0. (Cited on page 200.)

[115] Object Management Group. UML Version 2.1.2 Specification. http://www.omg.org/spec/UML/2.1.2/, 2008. Version 2.1.2. (Cited on page 188.)

[116] Object Management Group. UML Profile for MARTE: Modeling and Analisys of Real-Time Embedded Systems. http://www.omg.org/spec/MARTE/1.1, 2011. Version 1.1. (Cited on page 191.)

[117] Object Management Group. Object Constraint Language (OCL). http://www.omg.org/spec/OCL, 2011. Version 2.3 - Beta 2. (Cited on page 199.)

[118] Gøran Olsen and Jon Oldevik. Scenarios of Traceability in Model to Text Transformations. In *Model Driven Architecture-Foundations and Applications*, Lecture Notes in Computer Science. SINTEF, 2007. (Cited on page 109.)

[119] Open Source Modelica Consortium. The OpenModelica Project. http://www.openmodelica.org, 2011. (Cited on page 21.)

[120] OpenHMPP Consortium. OpenHMPP - Open Hybrid Manycore Parallel Programming. http://www.openhmpp.org, 2011. (Cited on page 24.)

[121] OpenMP Architecture Review Board. The OpenMP API specification for parallel programming. http://openmp.org/, 2011. (Cited on pages 2, 132, and 175.)

[122] S. V. Pennington and M. Berzins. New nag library software for first-order partial differential equations. *ACM Trans. Math. Softw.*, 20:63–99, March 1994. ISSN 0098-3500. doi: http://doi.acm.org/10.1145/174603.155272. URL http://doi.acm.org/10.1145/174603.155272. (Cited on page 1.)

[123] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. IEEE Press, 2004. (Cited on page 21.)

[124] Eric Piel. *Ordonnancement de systèmes parallèles temps-réel, De la modélisation à la mise en oeuvre par l'ingénierie dirigée par les modèles*. PhD thesis, Université des Sciences et Technologie de Lille, December 2007. in French. (Cited on page 39.)

[125] Python Software Foundation. Python Programming Language. http://www.python.org, 2011. (Cited on page 27.)

[126] Imran Rafiq Quadri. *MARTE based model driven design methodology for targeting dynamically reconfigurable FPGA based SoCs*. PhD

thesis, Université des Sciences et Technologie de Lille, April 2010. (Cited on page 40.)

[127] D. J. Quinlan, B. Miller, B. Philip, and M. Schordan. Treating a User-defined Parallel Library as a Domain-Specfiic Language. In *Proceedings of the 16th International Parallel and Distributed Processing*, Marriott Marina, USA, 2002. (Cited on page 23.)

[128] Fabio Remondino and Niclas Bõrlin. Polylib - a library of polyhedral functions. http://icps.u-strasbg.fr/polylib/ or http://www.irisa.fr/polylib. In *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences, Vol. XXXIV, Part 5/W16, H.-G. Maas and D. Schneider (Eds*, 2004. (Cited on page 106.)

[129] Calvin J. Ribbens, Layne T. Watson, and Colin Desa. Toward parallel mathematical software for elliptic partial differential equations. *ACM Trans. Math. Softw.*, 19:457–473, December 1993. ISSN 0098-3500. doi: http://doi.acm.org/10.1145/168173.168383. (Cited on page 145.)

[130] A. Wendell O. Rodrigues, Vincent Aranega, Anne Etien, Frédéric Guyomarc'h, and Jean-Luc Dekeyser. Enabling Traceability in an MDE Approach to Improve Performance of GPU Applications. Research Report RR-7720, INRIA, August 2011. URL `http://hal.inria.fr/inria-00617912`. (Cited on page 108.)

[131] Youcef Saad and Martin H. Schultz. Gmres: A generalized minimal residual algorithm for solving non-symmetric linear systems. *SIAM*, 7(3):856–869, 1986. doi: DOI:10.1137/0907058. (Cited on page 135.)

[132] Sadayappan, P. An Environment for High-Productivity High-Performance Computing using GPUs/Accelerators. Collaborative research, NSF, 2009. (Cited on page 30.)

[133] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2), February 2006. (Cited on page 187.)

[134] Sven-Bodo Scholz. With-Loop-Folding in SAC – Condensing Consecutive Array Operations. In *In Proceedings of the 9th International Workshop on Implementation of Functional Languages*, Scotland, 1998. Springer-Verlag. (Cited on page 29.)

[135] Robert S. Schreiber. Block Algorithms for Parallel Machines. In M. H. Schultz, editor, *Numerical Algorithms for Modern Parallel Computer Architectures*, number 13 in IMA Volumes In Mathematics and Its Applications, pages 197–207, 1988. (Cited on page 48.)

[136] Alfred Strey and Martin Bange. Performance Analysis of Intel's MMX and SSE: A Case Study. In *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, Euro-Par '01, pages 142–147, London, UK, 2001. Springer-Verlag. ISBN 3-540-42495-4. (Cited on page 171.)

[137] Julien Taillard. *Une approche orientée modèle pour la parallélisation d'un code de calcul éléments finis*. PhD thesis, Université des Sciences et Technologie de Lille, 2009. in French. (Cited on pages 10, 18, 39, and 132.)

[138] Julien Taillard, Frédéric Guyomarch, and Jean-Luc Dekeyser. A Graphical Framework for High Performance Computing using an MDE Approach. In *16th Euromicro International Conference on Parallel, Distributed and network-based Processing*, pages 165–173, Toulouse, France, February 2008. (Cited on page 36.)

[139] Goh Cheng Teng. Matrix Multiplication on GPU in Octave. Research report, Institute of High Performance Computing - A*STAR, February 2008. URL http://docs.ihpc.a-star.edu.sg. (Cited on page 27.)

[140] The Portland Group. PGI Accelerator Compilers. http://www.pgroup.com, 2011. (Cited on page 30.)

[141] T. P. Theodoulidis. Model of ferrite-cored probe for eddy current nondestructive evaluation. *Journal of Applied Physics*, pages 3071–3078, 2003. (Cited on page 130.)

[142] T. P. Theodoulidis. Analytical Model for Tilted Coils in Eddy-Current Nondestructive inspection. *IEEE Transactions on Magnetics*, pages 2447–2454, 2005. (Cited on page 130.)

[143] Didem Unat, Xing Cai, and Scott B. Baden. Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 214–224, New York, NY, USA, 2011. ACM. (Cited on pages 23 and 24.)

[144] F. Vazquez, G. Ortega, J. J. Fernandez, and E. M. Garzon. Improving the performance of the sparse matrix vector product with gpus. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, pages 1146–1151, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4108-2. doi: http://dx.doi.org/10.1109/CIT.2010.208. URL http://dx.doi.org/10.1109/CIT.2010.208. (Cited on page 145.)

[145] Min-You Wu, Wei Shu, and Jun Gu. Efficient Local Search for DAG Scheduling. *IEEE Trans. on Parallel and Distributed Systems*, 2001:617–627, 2001. (Cited on page 87.)

[146] Huafeng Yu. *Un Modéle Réactif Basé sur MARTE Dédié au Calcul Intensif à Parallélisme de Données : Transformation vers le Modèle Synchrone*. PhD thesis, Université des Sciences et Technologie de Lille, November 2008. in French. (Cited on page 39.)